AD NUMBER

ADB031358

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies only; Test and Evaluation; 26 MAY 1977. Other requests shall be referred to Ballistic Missile Defense Advanced technology Center, ATTN: ATC-P, PO Box 1500, Huntsville, AL 35807.

AUTHORITY

BMDATC notice dtd 6 Mar 1981

LEVEL II

HIGHER ORDER SOFTWARE, INC.
806 Massachusetts Avenue
Cambridge, Ma. 02139

(2)

TECHNICAL REPORT # 19

AXIOMATIC ANALYSIS

FINAL REPORT FOR

PERIOD AUGUST 1977 - SEPTEMBER 1978

SEPTEMBER 1978

Prepared for

Ballistic Missile Defense Advanced Technology Center

NOTICES

DISCLAIMERS

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government endorsement or approval of commercial produces or services referenced herein.

DISPOSITION

Destroy this report when it is no longer needed. Do not return it to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-19	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Axiomatic Analysis - Final Repurt for Period August 1977 - September 1978	5. TYPE OF REPORT & PERIOD COVERED: Final Report Aug. 1977 - Sept. 1978
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Higher Order Software, Inc.	8. CONTRACT OR GRANT NUMBER(s): DASG60-77-C-0155
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Higher Order Software, Inc. 806 Massachusetts Avenue Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Higher Order Software, Inc. Cambridge, MA 02139	12. REPORT DATE: September 1978
	13. NUMBER OF PAGES: 243
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Ballistic Missile Defense Advanced Technology Center Huntsville, AL	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution limited to U.S. Government Agencies only, Test and Evaluation, 26 May 1977. Other requests for this documant must be referred to Ballistic Missile Defense Advanced Technology Center, Attn: ATC-P, P.O. Box 1500, Huntsville, AL 35807.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Requirements, systems, system theory, system specification, HOS methodology, resource allocation, axioms, data types, functions, control structures, semantics, level, layer, natural language, category theory, arrows, commutative diagrams, duality, fuzzy sets, fuzzy logic, approximate reasoning, uncertainty, probability

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The basic concepts of the HOS systems theory and design methodology are developed. Notions like structure, data type, variable, value, function, tree, node, data structure, primitive operations, and universal primitive operations are elaborated and a distinction is drawn between a theory as a discovery procedure and as a set of constraints. Functions, algebras, and control maps are discussed in connection with data type specification, and the potential use of HOS in artificial intelligence and cognitive modelling is explored, including the

Over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

19. theory, notational frameworks, R-Nets, perspicuity, expressive power, comparison.

20. possible use of HOS as a model for lexical semantics. Some mathematical results relating the HOS primitive control structures to mathematical category theory are derived and the relevance of the theory of fuzzy sets and fuzzy logic to the BMD environment is examined. Four notational frameworks for specifying BMD-related systems and requirements are compared and evaluated along the dimensions of expressive power and perspicuity.

HIGHER ORDER SOFTWARE, INC.

806 Massachusetts Avenue

Cambridge, Ma. 02139

(15) DAAG60-77-C-0155

TECHNICAL REPORT # 19

(14) TR-19

(6) AXIOMATIC ANALYSIS.

(12) 234 p.

FINAL REPORT FOR

PERIOD AUGUST 1977 - SEPTEMBER 1978

(9) Final rept. Aug 77-Sep 78.

(11) SEPTEMBER 1978

Prepared for

Ballistic Missile Defense Advanced Technology Center

393 027

TABLE OF CONTENTS

PART 1

Introduction

INTRODUCTION

This is the final report of work done on BMD Contract DASG60-77-C-3049. The report is divided into eleven parts. The first five parts, after this introduction, deal respectively with preliminary concepts of system specification; foundations of axiomatic analysis; the relation between data types, functions, and control structures; software engineering and artificial intelligence; and the semantic structure of language. These five parts have appeared earlier as the first interim report. The next four parts deal respectively with the HOS methodology (Parts 7 and 8) and with fuzzy sets and fuzzy logic (Parts 9 and 10) and appeared earlier as the second interim report. The last two parts are new, appearing here for the first time. Part 11 compares four system specification methodologies and Part 12 applies some of these ideas to a real BMD problem.

Part 2 develops the basic concepts of HOS as a systems design methodology. The notions of structure, data type, variable, value, function, tree, node, data structure, primitive operations, and universal primitive operations are reviewed and formally characterized. The distinction between an object, its name, its mode of existence, its environment, when it happens, where it comes from, its representation, its implementation, and its execution is discussed and explicated. The notions of a system layer and a level of refinement are contrasted. Specific attention is paid to the characterization of a complete system specification and the problem of resource allocation in a system.

Part 3 develops the basic concepts of HOS as a systems theory. Two notions of axiom are discussed, one derived from empirical science and one from mathematics, and shown to play different roles within the HOS theory. The notions of a theory as a discovery procedure and as a set of constraints are compared and contrasted and the distinction is used to clarity the character of HOS.

Part 4 develops the basic concepts of HOS from the point of view of data-type specification, which in HOS is done algebraically. The notions of function, algebra, and control map are developed in connection with their use in specifying systems and found to provide a natural mathematical

characterization of the system concept. Some advantages of the HOS theory over other proposals for algebraic data-type specification that have appeared in the literature are discussed.

Part 5 explores the potential usefulness of HOS in the study and modelling of cognitive systems. Some conceptual and methodological problems in current work in artifical intelligence are pointed out and a way of remedying them by using HOS is suggested.

Part 6 represents a first step in implementing the program of research suggested and outlined in Part 5. A recent proposal in the linguistic literature that word meanings might be best represented as functions is examined in the light of HOS concepts of functional decomposition and found to lead to a natural algebraic model for the semantic lexicon of a language. The work reported in this part is of particular significance for BMD systems for three important reasons. First, many properties of BMD systems are properties they share, as systems, with other systems of seemingly very different kinds. The distinction between a theory of systems viewed as a discovery procedure and a theory of systems viewed as a set of constraints, for example, as discussed in Part 3, is derived from work in linguistics on syntactic systems. A lot can be learned that is relevant to BMD systems by analyzing the properties of other systems that are easier to deal with and that thus reveal their essential properties more readily. Second, a fully successful BMD system will require maximal automation in both its recognition and its response capabilities, including the ability to communicate with mechanical components of the system to the greatest extent possible in natural language. Syntax-based proposals for natural-language man-machine communication have fallen far short of expectation and need, and semantics-based methods would seem to provide much greater liklihood of success. If the semantic system of a language can in fact be described in exactly the same language and concepts--namely, HOS--as other components of BMD systems, then the entire BMD program can be greatly simplified and its efficiency significantly enhanced. Third, the specific example discussed, taken from the linguistic literature, involves exactly the kind of problem that it would be necessary to solve in a rigorous formalization of a BMD system. The example deals with the question of how best to characterize a number of data types that are particularly in need of for-

mal explication in a BMD system: EVENTS, ENTITIES, ACTIVITIES, LOCATIONS, and PATHS. An automated BMD response device, for example, would have to know precisely and unambiguously what constitutes an event requiring a response and what paths that response might involve. The discussion in this part can be viewed as a first attempt to develop answers to precisely these and related questions.

Parts 7 and 8 deal specifically with the HOS systems methodology, with Part 7 focusing on control structures and Part 8 focusing on data types; these two theoretical entities, along with functions, constitute the three basic units in terms of which any system can be specified in HOS. Part 7 examines the three HOS primitive control structures, usually defined in traditional set-theoretic terms, and asks what happens if the definitions are reformulated in terms of the newer mathematical theory of categories. This effort is in line with state-of-the-art thinking in mathematics itself, much of which is concerned precisely with such category-theoretic reformulations of set-theoretic notions. A number of revealing insights emerge form the analysis undertaken here, both in regard to the primitive control structures themselves and in regard to the very notion of "system." Part 8 focuses on data types from the user's point of view, outlining how one goes about constructing an algebraic specification for a data type in an HOS system. Three versions of data-type TIME that might be needed in BMD systems are presented and some theoretical point concerning consistency and completeness are discussed.

Parts 9 and 10 deal with fuzzy sets and fuzzy logic, branches of mathematics and logic that deal specifically with problems of vagueness and uncertainty, two pervasive characteristics of BMD-related situations and systems; in particular, the theory of possibility, based on fuzzy sets and logic, has been proposed as an alternative to probability theory as a way of dealing coherently with these two characteristics. Part 9 discussed fuzzy reasoning and its relevance to communication in systems; such as those that arise in the BMD environment. Part 10 surveys the various systems of fuzzy logic that have appeared in the literature and discusses their relative advantages and drawbacks.

Part II analyzes four notational frameworks for system specification and evaluates them along the dimensions of expressive power and perspicuity. The four frameworks include HOS control maps, R-Nets, commutative diagrams, and a modified version of the R-Net framework. Recommendations are made for the conditions under which the various frameworks might most fruitfully be used.

PART 2

Some Preliminaries on System Specification

by

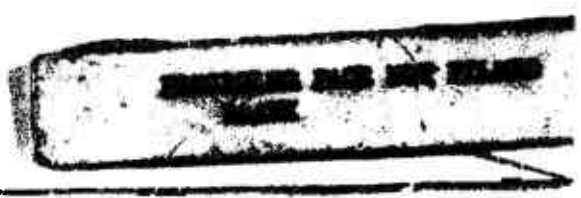
M. Hamilton and S. Zeldin

Some Preliminaries on System Specification

Design and Verification take place throughout a system development process. Design means to think. Verify means to think back. For each step of design there should be a counter-step of verification. At times, the process of design is one and the same as the process of verification. This occurs when certain design characteristics are included for the purpose of preventing unnecessary verification. In such a case, some types of verification requirements are designed out of the system. What is left is the second order verification which determines if unnecessary verification requirements with respect to a design have truly been eliminated, and then a need to verify only that which is truly part of the original intent of the design.

Many engineers talk about a desire to improve their own design techniques. These design techniques include those techniques for producing the design for a solution to a particular problem as well as the design of the process which will verify that solution. More often than not, these engineers appear to be talking about a different design process since they are involved in different types of systems or different phases of development within a given system. Actually, they are applying the same process (i.e., design) to a different "application."

From the point of view of a typical development process, design could be the process of going from a concept to a set of requirements, a set of requirements to a set of specifications, a set of specifications to a set of code or a set of code to a computer. In every one of these processes each designer considers himself to have the task of preparing his design to eventually reside in a computer or machine environment. One of the problems with this approach is that a designer either worries unnecessarily about design considerations not relevant to his own process or he might leave out design considerations thinking someone has already taken care of them or that someone will take care of them later. The important consideration for each designer is to worry about designs which he should worry about and only those designs he should worry about. Each designer goes through the same process, but each designer should be applying that process to a different phase of the overall application. Thus the inputs and outputs of his design process should be both unique and self-contained.



Other than a good deal of insight, a successful designer has necessary and sufficient knowledge about his problem, an understanding of the nature of a design process, an understanding of the nature of the reverse of a design process (the verification process), and a means to effectively perform a reliable and workable implementation of his design.

It is desirable for a designer to have a methodology to support him in a design process. However, although a methodology can support a designer, it can never replace him. A tool can be developed to replace some of his functions with respect to himself, the designer, and even to replace all of his functions on a particular project. However, the designer still has the prerogative to create new designs and to design new uses for the same tool or new tools for different uses.

There are many methodologies today whose intent is to provide standards and techniques to assist the engineer in the design and verification process [1]. The developers of these methodologies are all proponents of reliable designs. And, most methodologies advocate similar techniques towards this aim. For example, it is a commonly accepted idea that it is beneficial to produce a hierarchical breakdown of a given design in order to provide more manageable pieces to work with. And, there are variations between methodologies. Some emphasize a concentration on data flow as opposed to functional flow [2],[3],[4],[5]; others emphasize just the opposite [6],[7],[8]; others indicate that both functional and data flow are of equal importance [9]; others emphasize documentation standards [10],[11]; others emphasize graphical notations [12]; and still others emphasize semantic representation [13].

There are certainly positive aspects in many of these methodologies and, in particular, in what they are trying to obtain. To be effective, however, a methodology should have techniques and rules for the purpose of defining systems which are consistent and complete. But these techniques and rules are useful only if they are within themselves consistent and complete, both with respect to each other and to the systems to which they are being applied. With a complete and consistent methodology, a system can be defined formally so that all systems which communicate with a given system can understand that system in the same way.

In order to define a complete system, a methodology must have the mechanisms to define all of the relationships which exist in a system environment. This includes communication within and between systems and the resource allocation which provides for that communication. Thus not only must all data, data flow, function and functional flow be able to be explicitly defined, but the relationships (and control of the relationships) between data and data, between function and function, and between data and function must be able to be defined within any given system environment.

A methodology should have a mechanism to provide modularity in the formal sense. That is, any change should be able to be made locally and if a change is made, the result of that change should be able to be traced throughout both the system within which that change resides and throughout other systems communicating with that system.

A methodology should have the mechanisms to communicate and resource allocate a formally defined system in a manner which is transparent to the engineer, the user of these mechanisms. That is, even though the semantics of a system definition are formal, the syntax describing that system should be as flexible and as close to the "language" of the engineer as long as it contains the necessary information to be used in the system description.

We will discuss here some properties of the methodology of Higher Order Software (HOS) as they relate design and verification to the requirements we have set forth for methodologies in general.

An object is an existence of something. A system is an assemblage of objects united by some form of regular interaction or interdependence. The mechanism by which these objects are united can make the difference between a system which is reliable and one which is not. In HOS the objects of a system are united by a form of control where that form of control is determined by adherence to six axioms.

A system, itself, can be viewed as an object within another system. Objects of an HOS system can be described in terms of abstract control

structures [14] that relate members of algebraically defined data types [15],[5] or functionally defined data whose components are algebraically defined. When an object is viewed in terms of an abstract control structure, that object is in the state of doing. When an object is viewed as a member of a data type, that object is in a state of being.

HOS systems communicate in terms of data and functions. Functions can be in a being state or a doing state. Data can be in a being state or a doing state. Or, as Cushing puts it [16]:

> ...anything that can be...a datum, can also do, by serving as input to a function, and anything that can do, i.e., a function, can also be, since functions themselves make up a data type.

System designers often have the problem of knowing when they are finished. For example, it is not clear when a system definition has been decomposed as far as it should be or if it has been decomposed to a state of unnecessary detail. In HOS, a design that is finished is one which has been hierarchically decomposed into a complete system specification. In this case, all terminal nodes of a specification tree represent primitive operations on data types (Appendix I in [18]).

Sometimes we need to talk about the relationships between one system and another system. In order to talk about such relationships, it is necessary to understand the structure or organization of a system. In HOS the structure of a system can be made up of several systems where each system can be defined in terms of levels, layers, and instances of layers. A layer represents all performance passes of a system. An instance of a layer represents one performance pass of a system. A level represents a step of refinement within an instance of a layer of a system. A communication takes place between two systems if an instance of one system communicates with an instance of another system. Two systems are on the same layer if and only if an instance of one system always communicates with the same instance of the other system.

There are other relationships between systems other than that of communication between them as objects. A system, as an object, can communicate

with other layers of a system such as the definition, description, name, or environment. That same system, as an object, can also communicate with those systems on which another system is implemented. All of these relationships between one system and another system are defined by still another system. This process (or system) which prepares one system to relate to another system is called resource allocation.

In the process of resource allocation, it is possible to assign a name to an object, an object to a name, or replace an object by an object or a name by a name. An engineer provides resource allocations to objects in his system when he is designing the system. This process can take place manually or automatically. He also designs resource allocation functions for the system itself to perform when it is being executed. Sometimes it is necessary to perform a resource allocation to a system before it can communicate with another system. Sometimes it is necessary to perform a communication between two systems before a new step of resource allocation is determined.

Thus, for example, one system may need to compute an output value in order that such a value can be assigned to an input variable in another system. Another system may need to compute an output value in order that such a value can be assigned as a function to a name of a node within a structure in another system (this type of assignment is necessary for a reconfiguration of functions in real time). Another system may need to assign a function from one system as a value to an input variable inits own system. And still another system may need to assign an instance of a function as a value to an input variable in its own system.

When we think about an object it is often too easy to think about it in too limited a sense. The problem is that everyone involved with that same object has a different viewpoint of that object. And, it is for this very reason that one designer will define the same object differently than another designer. Each designer designs a system with his own implicit assumptions or preconceived notions that are collectively different from anyone else. Unfortunately, many of these inplicit assumptions are not usually made known until that object is finally executed on a computer. And sometimes, they arenever made known. At least a system

that is implemented on a computer has the advantage of the computer finding problems that designers of other non-computer problems would never find.

The only way to get around the problems of producing incomplete, ambiguous or redundant designs is to have the ability to define a system explicitly. In order to explicitly talk about an object, we need to understand what it is, its type (or characteristics), and its definition. (The definition of an object contains information about those objects controlled by that object as a controller. It does not include information about how that object itself is controlled.) To transfer an ojbect from one system specification to another, we would include all of the above in addition to the object, itself, and a description of that object.

The controller of an object allocates its name, how it exists (i.e., being or doing), its environment (i.e., its residence), when it happens, where it comes from, its representation, its implementation (i.e., what makes it happen), and its execution, which includes the relationships of one system, as an object, with another system as an object (i.e., its layer and level communication relationships) and the relationships of one system as an object with another system in order to prepare that system for communication (i.e., its layer and level resource allocation relationships).

An object as a controller resource allocates. An object as a function communicates.

Each object in a system specification has a special meaning to an engineer as to what it is. For example, objects such as functions, input and output values, and input and output variables are commonly referred by system designers. In HOS, we include these kinds of objects with very specific meanings, and we have other kinds of objects including those which are available as a convenient means of abstracting (or modularizing) system specifications. When we talk about an object, it is very helpful to say what it is, for in doing so we immediately know a lot more about its properties. For a given specification, an object in HOS could be, for example, a structure, an operation, a function, a data type, a

variable, a value a tree, a data structure, or a primitive operation. An example of objects that can be used in a system specification is shown in Figure 1. (A definition for these objects is given in Appendix I and Appendix II of [18].)

We can tell the type of an object by referring it to a set of values whose characteristics we have defined. A type is a set of values defined by a set of axioms. Each axiom is a true statement about control structures. Thus if

```
y = A(x);
```

we can say

```
Where (y,x) are INTEGERS,
      A is a constant FUNCTION;
```

in addition, if

```
z = C(b);
```

we might say

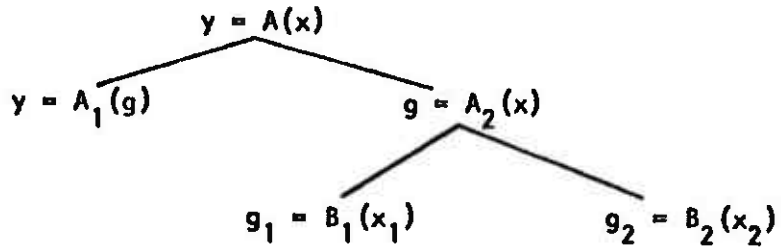
```
b is an A,
C is a FUNCTION,
z is a MATRIX;
```

To understand the meaning of a system, we define each of its objects. For example, one definition of an instance of E function, E, is

```
y = E(x);
Where x is an INTEGER,
      y is a NATURAL;
```

One definition of E as data is

```
Where E is a FUNCTION;
```



STRUCTURE:	the control relationships of the functions in System A
DATA TYPE:	a set of values characterized by a set of primitive operations
VARIABLE:	a name of an object "x" is an input variable of A "y" is an output variable of A
VALUE:	object being names x is a value of data type integer (e.g., 1,2,3,...) y is a value of data type integer
FUNCTION:	mapping between input values and output values where those values are represented by particular variables A, A_1, A_2, B_1 and B_2 are functions of System A
TREE:	geographical representation of a system (environment of A)
NODE:	location on tree. Function A resides at the root node of the control map for System of A
DATA STRUCTURE:	a set of variables whose values collectively represent the same value as a variable. In System A (x_1, x_2) is a data structure of x.
PRIMITIVE OPERATIONS:	one of a set of operations which characterize a particular data type. A_1, B_1 and B_2 are primitive operations for System A
UNIVERSAL PRIMITIVE OPERATIONS:	operations in which the operands are represented by variables whose values are of some type

Figure 1

We use a description to talk about the definition of a system. One description of an instance of function E is "y = E(x)". This description can be replaced by other descriptions. It could be replaced by ones described in an assembly language, specification language, or a higher order language. In HOS we describe definitions with AXES or AXES based statements (Appendix II in [18]).

We need to have a name for an object in order to mention that object [17]. A name itself can be an object with respect to its name. To talk about E, for example, we use the name of E, "E". E can be controlled by mentioning the name of E. Thus "E" controls E in that the mentioning of E makes E happen, and "E" can be controlled by naming "E". Thus "'E'" controls "E". To give a name of E is to resource allocate "E" to E. In order to invoke E we need to have an object, E_c, which mentions the name of E. E, "E", "'E'" and E_c, which invokes E, all reside in some environment. In order to control E by E_c we must also resource allocate to E_c a name in order to talk about the controller E_c. E_c and its name must also reside in some environment. In Figure 2, A, A_1, and A_2 are nodes which locate positions on the control map; F, F_1 and F_2 are functions in System E. In order to obtain F on the control map, we locate F by F = Object("F"). F, F_1 and F_2 are resource allocated by the very fact that they have names. F, as a controller, relates to the invocation of F_1 and F_2 by mentioning their names which is, in essence, also a process of resource allocating the names "F_1" and "F_2" to nodes A_1 and A_2. In the mentioning of "F_1" and "F_2", F as a function is replaced by functions F_1 and F_2. Similarly, in the case, for example, where we have an instance of the function F, defined as y = F(x), we are concerned with "y" and "x", in addition to "F", as well as where these names reside.

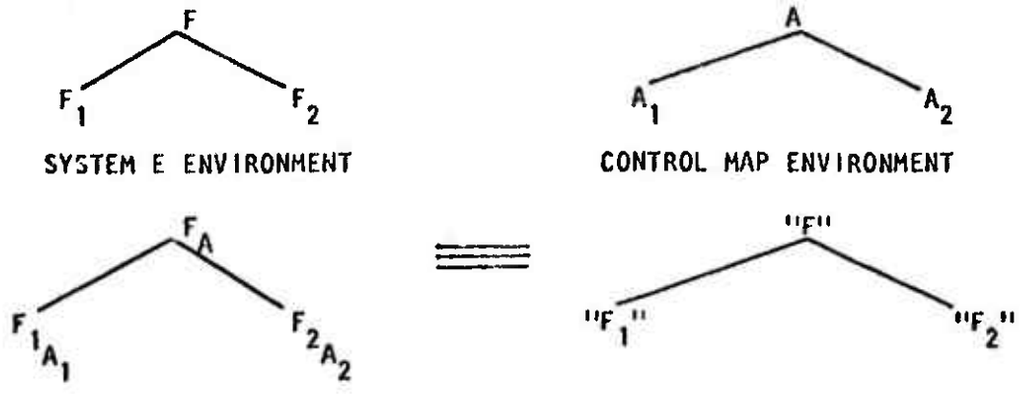


Figure 2: System E on the Control Map

When we relate to an object, it si important to know how it exists. In the definitions above, when E is viewed in terms of a control structure, E and x and y are viewed as doing. When E is viewed in terms of a type, E and x and y are viewed as being.

When one system relates to another system, it must know, among other things, the environment within which that system resides. This includes where that system is located and when that system exists.

In the case of Function E, we could say that the location of "y = E(x)" in on the line on this page in this collection of papers. For another Function A, we might say that its description, "y = A(x)" is located at node 12 on the control map. Or we might say that "y = A(x)" is located at Register 10 on $Computer_x$ or at $Record_A$ on $file_B$. Similarly with the definition of A, we could say y = A(x) is located at $BLOCK_{100\ to\ 200}$ on Computer. With respect to function , B, and time, t, we could say y = A(x) at t or y = A(x) after B or y = A(x) in t or y = Z(x) before t.

An example environment might be defined as follows:

$$y = A(z)_{at\ t\ at\ x}$$

Where t *is a* TIME,
z *is a* LOCATION

or if the type information about A, t and z were contained on, for example, a file, we could simply say

$$A_{at\ t\ at\ x}$$

The representation of an object might vary depending on its environment. One representation of A, A's controller, their names, names of names, etc. could exist in a computer. In one computer, a rational number could be represented as a single-precision number, and on another computer, it could be represented as a double-precision number. In an HOS system specification, we often talk about these objects within the environment of a control map. In an HOS implementation, the nodes of a control map can be replaced by locations of a computer, the names of the nodes by names of instructions or names of locations in the computer. Thus

if a specification is described within a control map environment, it can be transferred to the computer environment as part of the implementation process.

In order to implement a system (or make that system happen), we define that system, the environment of that system and the system that will execute that system. That system which executes another system is called a machine (Appendix III in [18]).

One of the most difficult problems facing system designers is that of resource allocating a design to execute on a particular machine. More subtle problems appear when we attempt to redesign our system to fit a machine. Once having done so, we can no longer understand our original design, since the resulting complexity hides the original intent with camouflages of implementation. We not only cannot trace input and output throughout our system design, but we are no longer even sure which input and output is relevant to our original problem. To try to change such a system with a new system requirement is a presumptuous notion.

Input and output can be traced throughout a given AXES system definition. Also data flow can be separated with respect to different layers of implementation. These features allow us to look at system interfaces which are relevant at the time they are relevant. We often hear the complaint from system designers about the problem of attempting to design a system "top-down" and then being forced to add details of implementation on a lower level of the top-down design when resource allocation is addressed. This forces an iteration of the design process in order to incorporate the extra details of implementation back at the top and down through the top-down design to maintain consistency of the overall design. (Some designers merely add that extra detail without worrying about it because they don't know what to do with it, except to say it came from some other system; but then they lose track of its influence on their own system and the influence of their own system on other systems.)

In order to resource allocate a given system to a particular machine, it is necessary to define both the system and the machine in such a way that a change to one won't affect the other. It is also necessary to define

the machine in such a way that the users of the same machine only affect each other when they are explicitly deifned to affect each other.

Whenever we want to separate a system from the specification of its execution, we specify one layer for the system and another layer for the machine that runs that system. Likewise, that same machine can be looked upon as a system with respect to the machine which executes it. To implement System R, for example, a user might want to "run" R on the OS system. He could express such an implementation as

R *on* OS;

or he could express a more detailed implementation as

R at A on B *on* OS at m on p'

Where A *is a* RECORD,
B *is a* FILE,
m *is a* LOCATION,
P *is a* ROM;

In AXES these or similar statements can be used as long as the syntax and the semantics of the syntax are defined using AXES. The process of implementation continues until we arrive at the layer of a primitive machine. Consider potential resource allocation steps for a system:

Determine system: (1)
S *is* System;

Determine I/O: (2)
Where S *on* (x_1,x_2);

Determine algorithms: (3)
(x_1,x_2)
Where (x_1,x_2) *on* E;

Determine OS: (4)
$x_2 = E(x_1)$
Where E *on* Executive;

Determine E computer: (5)
$Estate_n = Executive(Estate_1)$
Where Executive *on* $computer_x$

Computer: (6)
$Cstate_n = Computer_x(Cstate_1)$;

Because each layer is defined as a control hierarchy, each machine that runs a system is able to use that system description as data and therefore has access to information about that system's ordering relationships as well as that system's data relationships. With respect to the six different layers for System S shown above, any one of these layers can be replaced by another implementation layer. If we want to move E to another operating system, we change Step 4 to name a new OS for E and replace Steps 5 and 6. If we wish to design a new algorithm for the same computer, we change Step 3 and replace E with a new algorithm and then replace Steps 4, 5, and 6. If we want to move to a new computer and our operating system is independent of its computer (as it ideally would be), we can change Step 5 to refer to another computer and we then can replace Step 6.

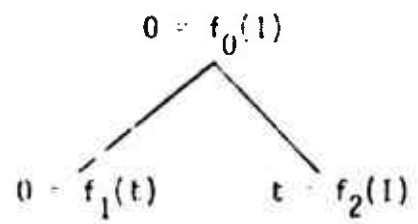
The execution of a system is a continuous process of resource allocating system objects for the purpose of communication and processing the communication between these objects, where that communication determines the next step of resource allocation. The start of an execution of an object is like the process of one object mentioning another object by first calling its name and then operating on the object itself.

By executing a system we realize the implementation of a system. An engineer can manually execute a system with the aid of a pencil and paper. One, however, more commonly thinks of computers as executing a system. In both cases, this execution process is a dynamic one. In the dynamic process we execute a system instance by instance. Each instance is a performance pass of the system.

Sometimes a system is viewed statically. In this case an engineer or a computer observes the description of a system by eyeballing statements or instructions. Examples of such static views are those performed by a compiler or an interpreter. An OS could view a system in both its static and dynamic states.

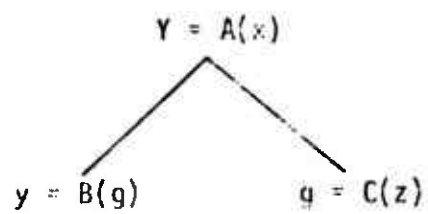
When we wish to define a set of systems to communicate with each other, we define a system whose purpose is to control their communication.

In order to define such a system, we have available a set of rules which will help us determine how to properly connect systems. As an example, consider two systems which are to be resource allocated to directly communicate with each other. A primitive control structure exists which provides the rules for properly defining two such systems, i.e., one of which is dependent on the other. Since we are in the specification stage, we will choose to have the system reside on the control map, which in our case resides on a piece of paper. It could just as well reside on a graphics device. Here the primitive control structure, composition (Figure 3a), is used as a model for organizing our functions. That system which controls their communication is their controller.



WHERE f_0, f_1, f_2 ARE FUNCTIONS
I,t,O ARE OF SOME TYPE

(A) Primitive Control Structure, Composition



WHERE A IS A CONSTANT FUNCTION
x,y,g ARE INTEGERS;

(B) Use of Composition

Figure 3
An Example of the Resource Allocation
of Systems Communicating on the Same Layer

We can tell from this control structure that any function can be "plugged into" f_0, f_1, and f_2 as long as it follows the rules of composition since f_0, f_1 and f_2 are defined as functions (Appendix II in [18]). A user of the composition control structure could plug System A into the structure as follows:

Where A *on* f_0, A_1 *on* f_1, A_2 *on* f_2;

Or, in Figure 3b, we have another example of the use of the structure in Figure 3a. Here, A, B, and C are values for f_0, f_1 and f_2 respectively.

In this use of the composition control structure, A controls B and C to communicate with each other on the same layer. A, as a controller, can be thought of as giving control commands where

C *when* x

B *when* g *from* C.

(Notice that B is viewed as a being and C is viewed as both being and doing by A.)

Since A is a constant function, it follows also that B and C are constant functions. Thus B and C are on the same layer of communication since both always relate to the same instance with respect to A.

The control relationship of A, B, and C are defined by composition, which is defined as an AXES based *STRUCTURE, Join* (Appendix II in [18]). Other control structures exist for the purpose of providing a controlled communication between systems. These include the other primitives, set partition (decision making) and class partition (parallel processing or independent functions). From a combination of primitives, we can form more abstract control structures (e.g., recursive functions). The rules for the control relationships of the primitive control structures are described in Appendix I of [18]. Similar rules exist for every defined AXES control structure.

The syntax that is associated with such *STRUCTURES* is used manually by a designer to define system specifications. The computer can automatically do the same thing with a *STRUCTURE* by writing the equivalent commands

into a register instand of on piece of paper. One way of specifying such a command is through the AXES *on* statement. These structures provide the mechanism for systems on different layers to communicate with each other in that the structure provides a means for an instance of one system to communicate with an instance of another system by simply adhering to the relationships of the structure in real time.

We use such a concept for the communication between functions in an asynchronous environment. For example, a control structure which can be used to define an interrupt is

STRUCTURE: $(y_1,y_2) = Xch(x_1,x_2);$

Where y_1,y_2,x_1,x_2 *are of some type;*

$y_1 = id_2^2(x_1,x_2)$ *Coinclude* $y_2 = id_1^2(x_1,x_2);$

SYNTAX: $(y_1,y_2) = Xch(x_1,x_2);$

END Xch;

Here the universal operations, id_2^2 and id_1^2, themselves defined in AXES to select a value from a set of values, as well as the *Coinclude*, which is one of the structures defined so that we can access the same value more than once, is used to define Xch.*

*We named this structure after the XCH instruction in the Apollo Guidance Computer (AGC), which was used to perform an interrupt.

REFERENCES

[1] Ramamoorthy, C.V. and So, H.H. "Appendix A to Requirements Engineering Research Recommendations," Software Requirements and Specifications: Status and Perspectives, August 1977.

[2] Jackson, M.A. Principles of Program Design. Academic Press, N.Y., 1975.

[3] Bridge, R.F. and Thompson, E.W. "A Module Interface Specification Language," Information Systems Research Laboratory, University of Texas at Austin, Technical Report No. 163, Dec. 1974.

[4] Guttag, J., et al. "The Design of Data Structure Specifications," Proc. 2nd International Conference on Software Engineering, Oct. 1976.

[5] Robinson, L. and Holt, R.C. "Formal Specifications for Solutions to Synchronization Problems," Computer Science Group, Stanford Research Institute, 1975.

[6] Computer Sciences Corporation, "A User Guide to the Threads Management System," Nov. 1973.

[7] Alford, M.W. "R-Nets: A Graph Model for Real-Time Software Requirements," Proc. MRI Symposium on Computer Software Engineering, April 1976.

[8] Davis, C.G. and Vick, C.R. "The Software Development System," Proc. 2nd International Conference on Software Engineering, Oct. 1976.

[9] Hughes Aircraft Company. "1975 IR&D Structured Design Methodology, Vol. II: Structured Design," FR 76-17-289, 1975.

[10] Teichroew, D. and Hershey, E.A. III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol SE-3, No. 1, Jan. 1977.

[11] IBM. "HIPO: Design Aid and Documentation Tool," IBM, SR20-9413-0, 1973.

[12] Ross, D. "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, Jan. 1977.

[13] Wilson, M.L. "The Information Automat Approach to Design and Implementation of Computer-Based Systems," IBM, Report IBM-FSD, June 1975.

[14] Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software," IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, March 1976.

[15] Hamilton, M. and Zeldin, S. "AXES Syntax Description," Technical Report No. 4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

[16] Cushing, S. "The Software Security Problem and How to S-lve It," Technical Report No. 6, Revision 1. Higher Order Software, Inc., Cambridge, MA, July 1977.

[17] Searle, J.R. "Review of J.M. Sadock, Toward a Linguistic Theory of Speech Acts," Language 52, 1976.

[18] Hamilton, M. and Zeldin, S. "Verification of an Axiomatic Requirements Specification" presented at the AIAA Computers in Aerospace Conference, Los Angeles, CA, Oct. 1977.

PART 3

Foundations of Axiomatic Analysis

by
S. Cushing

Foundations of Axiomatic Analysis

1.0 AXIOMS AND ALGEBRAS IN GENERAL

One of the most powerful and useful tools nvailable to science ls the general process of abstraction. From a large collection of diverse facts, i.e., true descriptions of data and phenomena that have been derived from experience and reasoning, the scientist tries to extract a set of unifying generalizations from which all of those facts can be logically deduced. The ancient Babylonians were quite familiar with what we call the Pythagorean theorem, for example, and similar principles thnt they derived from surveying and other forms of empirical measurement [Beh74], and it was only much later, under the influence of philosophers like Plato, that Greek mathematicians, such as Euclid, managed to show that all of those principles could be logically derived from a tiny subset, which they called "postulates" or "axioms." Similarly, Newton distilled all of the empirical physical facts and observations that had been collected by his predecessors and contemporaries and showed that all of those facts could be derived as "theorems," and thus made intelligible from three basic principles, which he called "laws of motion." For almost three hundred years, Newton's "laws" served as the axioms of physics, just as Euclid's principles had been serving for a much longer time as the axloms of geometry.

During the past century, a more formalized notion of axiom has been developed by mathematicians, leading to the emergence of abstract branches of mathematics like group theory, linear algebra, and topology, which although they may ultimately have their origins in empirical science, have no direct or obvious connection to the real world. A group, vector space, or topology is any set of objects that satisfy the relevant set of axioms. Some groups may consist of numbers, others of functlons, and still others of rotations in the plane, but they are all groups because they all satisfy the relevant axioms. The axioms specify the basic structure of the sets as groups, while any other fact about them that is relevant to their being groups can be derived from the axioms as theorems.

Formally, this leads us to the notion of an (abstract) algebra [Beh74], [Bir70], [Gut75], [Ham76a]. An algebra is an ordered pair $[\Sigma,\omega]$, where

Σ is a non-empty class of non-empty sets, and ω is a non-empty class of operations on the grandmembers (i.e., members of members) of Σ. The members of Σ are called categories of the algebra, and the members of ω are called the primitive operations of the algebra. A particular algebra can be specified by giving a category specification, an operation specification, and an axiom specification. A category specification lists o. defines the members of Σ An operation specification gives the domains and ranges of the members of ω as Cartesian products of the members of Σ. An axiom specification is a non-empty set of formal statements that characterize the interactive behavior of the members of ω and the grandmembers of Σ. Algebras can be classified according to the constraints that we choose to put on one or more of their category, operation, or axiom specifications. An algebra $[\Sigma,\omega]$ is said to be homogeneous, if Σ contains exactly one non-empty member, while an algebra which is not homogeneous is said to be heterogeneous. Probably the most familiar kind of homogeneous algebra is the group and the most familiar heterogeneous algebras are the vector spaces.

Given the formal category, operation, and axiom specification of an algebra, we are free to implement the algebra any way we want, as long as we guarantee that the implementations of the categories and primitive operations behave in reality as the axioms say they should. We can then go ahead and prove all sorts of things about the algebra as theorems that we might never have suspected ahead of time. Since we prove these theorems formally from the axioms, without worrying about how our algebra might be implemented, the theorems that result apply equally well to every implementation, regardless of how different these implementations may be in other respects. By staying strictly within the formalism we automatically guarantee that we are talking only about those aspects of a situation that we really want to be talking about, i.e., those aspects that we have formalized in our axioms. This prevents bringing in undesired information about particular implementations that would inadvertently rule out other implementations that might later turn out to be desirable.

2.0 AXIOMS AND ALGEBRAS IN HOS

Higher Order Software (HOS) makes use of the ideas in Section 1 in two distinct, though related, ways, reflecting the two general kinds of entity that can exist in any computer system [Lin76], [Wal75], [Cus77]. HOS recognizes that there are essentially two modes of existence in the world, that of *being* and that of *doing*, and that everything generally manifests both modes at once. A given thing can either *be* or *do* and, in general, will both *be* and *do* at the same time. This dichotomy reflects the related bifurcation between *being* and *becoming*. If there is something that is *doing*, then there is something (perhaps the same thing) that is *being done to*, and this latter thing is therefore *becoming*. Again, in general, anything that is *doing* is also *being done to* and so is itself *becoming*, as well as *being*.

This enables us to understand the important relationship between *constancy* and *change*. If we remove the front element from a queue, for example, we still have the *same* queue, with one element removed, but we also have a *different* queue, i.e., the one that differs from the original one in exactly that element. The queue can still *be* the *same* queue, even though it has *become* a *different* queue, and we are free to choose whichever of these aspects of the situation fits our needs for any particular problem. We can also say the queue has changed its *state*, stipulating that the queue itself has not changed, but then it is the *states* that are *being* or *becoming*, so the same dichotomy emerges again on a higher level of abstraction.

HOS expresses the distinction between *being* and *doing* in terms of the familiar notions of *data* and *function* (or *operation*), and it does this in a completely formal way. Anything that can *be* can be represented as a member of a *data type*, and anything that can *do* can be represented as a *function*. As we would expect from a correct formulation, anything that can *be*, i.e., a datum, can also *do*, by serving as input to a function, and anything that can *do*, i.e., a function, can also *be*, since functions themselves make up a data type.

For example, if a datum x is mapped by functions f_1, f_2, f_3, f_4, f_5 onto data y_1, y_2, y_3, y_4, y_5, respectively, then x itself can be viewed as a function that maps the data f_1, f_2, f_3, f_4, f_5 onto y_1, y_2, y_3, y_4, y_5. Functions themselves can be data, in other words, and data can be functions, depending on the requirements of the particular problem we are working on. If FXY is the subset of data type FUNCTION whose members map data type X into data type Y, then X is the subset of FUNCTION that maps FXY into Y. Both interpretations are correct, in general, and which one we choose depends on what we need for a specific problem.

Again in accordance with the fundamental dichotomy, although data and functions are distinct components of systems, they are at the same time inseparable from each other, because each is characterized formally in terms of the other. A function consists of an input data type, called its domain, an output data type, called its range, and a correspondence, called its mapping, between the members of its domain and those of its range; a function can be characterized, therefore, as an ordered triple (Domain, Range, Mapping), where the components are as we have just stated. A data type consists of a set of objects, called its members, and a set of functions, called its primitive operations, which are specified by giving their domains and ranges, at least one of which for each primitive operation must include the data type's own set of members, and a description of the way their mappings interact with one another and, perhaps, with those of other functions; a data type can thus also be characterized as an ordered triple, this time (Set, DR, Axioms), where Set is the set of its members, DR is a statement of the domains and ranges of its primitive operations, and Axioms is a description of the interactive behavior of the mappings of the primitive operations. A data type is thus characterized as an algebra, as we defined this notion in Section 1.0.

An example of an IIOS data-type specification, namely, type STACK, is given in Figure 1, written in the IIOS specification language, AXES. The category specification is given by the WHERE statements, which tell

```
DATA TYPE:  STACK;
PRIMITIVE OPERATIONS:
stack1 = Push(stack2, integer1);
stack1 = Pop(stack2);
integer1 = Top(stack1);

AXIOMS:
     WHERE Newstack IS A CONSTANT STACK;
     WHERE s IS A STACK;
     WHERE i IS AN INTEGER:
Top(Newstack) = REJECT;
Top(Push(s,i)) = i;
Pop(Newstack) = REJECT;
Pop(Push(s,i)) = s;
END STACK;
```

Figure 1

HOS/AXES Data Type Stack

us that E contains two members, the set of stacks and the set of integers. The operation specification of the algebra is given under the heading "PRIMITIVE OPERATIONS" and says that Ω consists of three members: Push, which maps stacks and integers onto stacks; Pop, which maps stacks onto stacks; and Top, which maps stacks onto integers. The axiom specification of the algebra appears after the WHERE statements and contains four axioms stating the behavior required of a set of objects in order for them to qualify as a set of stacks: the Newstack has no top; the top of the stack that results from pushing an integer onto a stack is that integer; the Newstack cannot be popped; popping the stack that results from pushing an integer onto a stack is that original stack. The existence of a stack with the specific properties attributed to Newstack in the first and third axioms is specified in the first WHERE statement. REJECT is assumed to be a member of every data type. It is invisible to universal quantification and it is produced as output by a function when no genuine output is produced. This specification is completely self-contained, saying exactly what we intend to mean by the word "stack," and it is entirely implementation-free. Any set of objects can qualify as stacks, as long as the primitive operations we want to perform on them satisfy the axioms of the algebra.

An HOS data-type specification characterizes instances of the two fundamental modes of existence in terms of each other. A kind of data (being) and a kind of function (doing) are specified as behaving towards each other as indicated by the axioms of an algebra. Given such an algebra, however, we might want to specify new functions that also operate on that kind of data. Furthermore, we might want to give the data of that type a data structure in terms of data of a different data type. Both of these aims are achieved in HOS by the specification of decomposition trees, also called control maps, which must themselves satisfy a certain set of axioms in order to be well-formed. These trees can also be viewed from the opposite direction, as decomposing more complicated functions into less complicated functions, and finally into the primitive operations of data types. Given a system that involves certain data types, the

function the system performs can be decomposed intu a tree structure whose nodes are functions nnd whose terminal nodes, in particular, nre primitive operntions of the dnta types, where the collective effort of the functions at the terminal nodes is the same as that of the system as a whole. Such tree structures are not intended to provide definitions of kinds of objects, which are provided hy the data-type specification (algebras), but represent system decompusitions into subsystems. An example of such a decomposition tree, for the function $y = \frac{a+b}{c-d}$, is shown in Figure 2. The domain and range of the decompused function can he determined by the typed variables that represent Inputs and outputs and by the primitive operations that appear at the terminal nodes. The tree itself is precisely what gives the mapping of the decomposed function, hy showing how that mapping gets accomplished in terms of the collective behavior of the independently characterized prinitive operations.

The key to the usefulness of these decomposition trees lies in the six HOS axioms, iisted in Figure 3. It is these axioms, in fact, and their consequences, of course, that make HOS HOS. While HOS can specify any system that can be specified, the specification must be in accordance with these axioms or the system may be incomplete or unreliable. Any software system, in particular, that is specified in accordance with these axioms is automatically guaranteed to be reliable, in the sense that no data or timing conflicts can ever occur [Ham76b], and secure, in the sense that information flows only upward [Cus77]. Formally, the axioms tell us that a well-formed HOS tree ls always equivalent to a tree in which every node is occupied by one of the three primitive controi structures, shown in Figure 4. Abstract control structures, defined in terms of the primitives may also appear in well-formed trees, and, conversely, any control structure, i.e., conflguration of parent and offspring nodes, can appear in a well-formed tree as long as it can itself be decomposed into the primitives.

Such an HOS tree can be interpreted either as decomposing a function into primitive operations or as building up a function out of primitive operations. Which interpretation we choose for a particular tree depends,

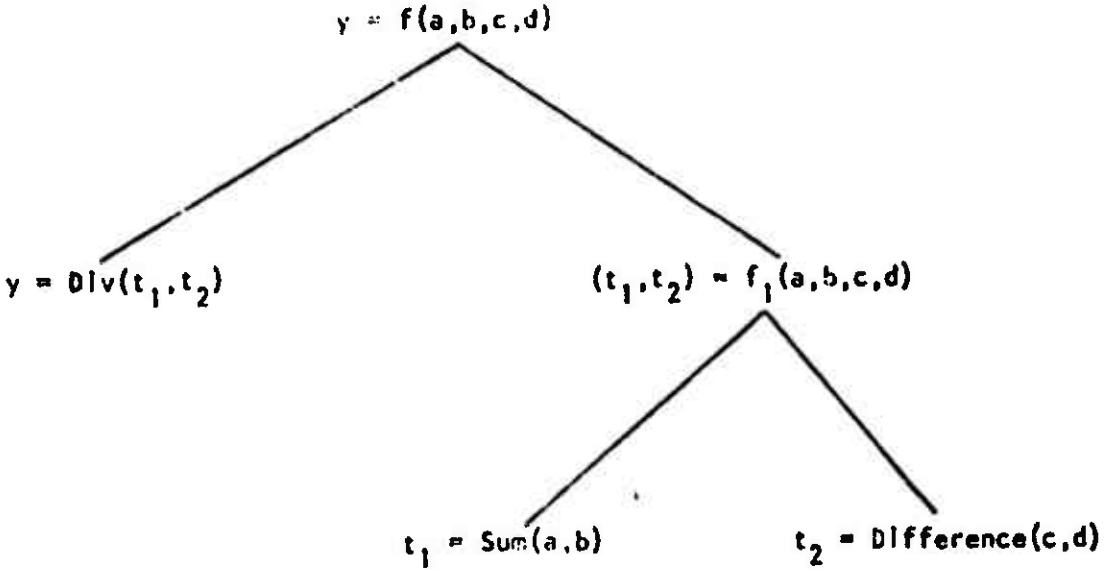


Figure 2

HCS Tree for Function $y = \frac{a+b}{c-d}$

DEFINITION: Invocation *provides for the ability to perform a function.*

AXIOM 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate lower level.

DEFINITION: Responsibility *provides for the ability of a module to produce correct output values.*

AXIOM 2: A given module controls the responsibility for elements of its own and only its own output space.

DEFINITION: *An* output access right *provides for the ability to locate a variable, and once it is located, the ability to give a value to the located variable.*

AXIOM 3: A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate lower-level function.

DEFINITION: *An* input access right *provides for the ability to locate a variable, and once it is located, the ability to reference the value of that variable.*

AXIOM 4: A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate lower-level function.

DEFINITION: Rejection *provides for the ability to recognize an improper input element in that, if a given input element is not acceptable, null output is produced.*

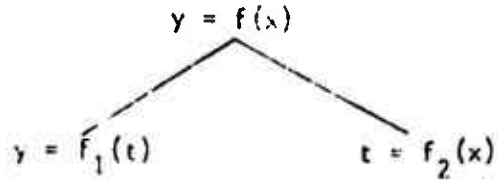
AXIOM 5: A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering *provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of the said elements precedes the other said element.*

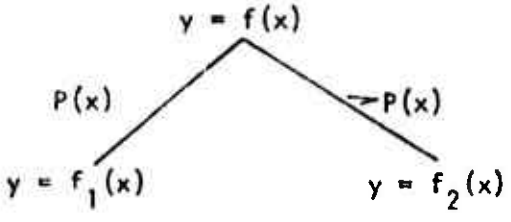
AXIOM 6: A given module controls the ordering of each tree for its immediate, and only its immediate, lower level.

Figure 3

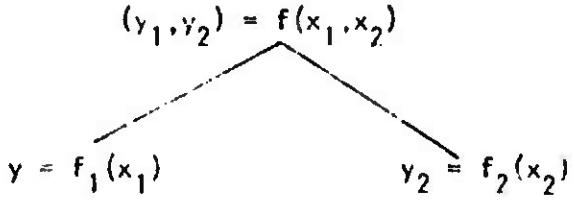
The Axioms of HOS



COMPOSITION



SET PARTITION



CLASS PARTITION

Figure 4

The Three Primitive Control Structures of HOS

as usual, on the use we want to make of it. Under either interpretation of such a tree, however, what we end up with is a specification of the function at its root node that is genuinely non-procedural, i.e., non-algorithmic, and entirely free of implementation considerations. The tree provides a complete and explicit account of what functional mapping the function performs and how that mapping is collectively carried out on the types involved by their primitive operations. Everything is clearly spelled out in terms of the hierarchical organization of functional mappings, and this--no more, no less--is exactly what we require of an adequate specification methodology; this, in turn, is what HOS is intended to provide.

To clarify this latter point somewhat, suppose we have a register whose positions are filled with integers. Obviously, there is a big difference between an implemented register and the integers it contains, and thus between changing the state of the register and taking one of those integers as a value. From the point of view of specification, however, a register is every bit as much of an abstraction as an integer. The two abstractions differ, moreover, only in the interactive behavior of the primitive operations that are used to characterize their data types, as this behavior is specified in the axioms of the respective type. From the point of view of specification, therefore, changing the state of an implemented register amounts simply to producing a new abstract register as a value. If we take a register and remove its last element, for example, we get a new register that is identical to the original register except that it lacks the original register's last element. This may not be what happens in implementation, but it is the logic of the situation, and that is what specification is really all about. Note, by the way, that this is just another way of looking at what we said about queues in the second paragraph of this section.

There is a subtle but important difference between the two uses of axioms in HOS, which we can illustrate most clearly, perhaps, by means of analogy. Data-type axioms are used within the HOS theory and are similar to mathematical axioms, such as those of group theory, for example. Given the general theory of HOS, we can choose arbitrarily, for whatever reason

we want, to specify any kind of object at all as an algebra and then determine the relevant axioms to include in our specification. The algebra that we construct represents (perhaps infinitely) many possible implementations, just as the group axioms represent (infinitely) many implementntions of the group notion. If we let "G" he a variahle that takes groups as values and "g" a variahle that takes members of G as values, then to say "s is a STACK" corresponds to the statement "g is a memher of G," whereas "S is an implementation of STACK" corresponds to "G is a group." Mathematicians do not usually talk in terms of individual groups heing implementations of the general notion (algebrn) GROUP, but this is what they really mean and we make it explicit in HOS. Formally, we say that a STACK exists on many possible layers, each of which is an implementation of data-type STACK, as discussed in [Ham76c, 77].

In contrast to the situation of data-type axioms, the six HOS axioms (Figure 3), are not freely chosen for particular uses within the HOS theory, hut constitute part of the definition of that theory. The HOS axioms are analogous not to strictly mathematical axioms like those of group theory, hut more to the laws of motion of Newtonian physics, whose axiomatic character we discussed in Section 1.0. Newton's laws are not sufficient by themselves to characterize the basic properties of the physical universe (as knnwn in his time), but presuppose and complement a formalization of what we would mean by "a universe" in the first place. The notion "a universe" can be characterized [Rya75] in terms of mathematicnl notions like differentiable manifold, vector field, and others, each of which involves axioms of its own. Once we get this notion straight, then we can add various other constraints as further axioms to characterize different kinds of universe--the Newtonian universe, the Einsteinian universe, the Brans-Dicke universe, etc.--and it becomes an empirical question which of these different theories of the universe really corresponds to the actual universe. The Newtonian universe was sufficient to account for all known facts up until the end of the last century, when the need for a new model, and thus a different set of supplementary axioms (constraints) beyond the strictly mathematical ones became apparent.

A completely formalized account of HOS, similarly, would have to include strictly mathematical axioms, such as those that are standardly used to characterize notions like tree, function, and so on, just as a completely formalized account of Newtonian physics requires axioms for things like differentiable manifolds. These axioms would provide what we might want to call "a systems theory," analogous to the formal notion "a universe." What we have called the six HOS axioms then serve as additional constraints that narrow down this notion to the HOS systems theory, just as Newton's three "laws" narrow down the notion "a universe" specifically to the Newtonian one. Since systems theory is a form of engineering, however, rather than an empirical science, the criterion for acceptability is not empirical accuracy, as it is for physics, but goals like reliability and security, which are automatically guaranteed by the six axioms.

Another difference also follows from this latter fact. In physics, the facts generally tend to underdetermine the theory, in the sense that there is more than one model of the universe that fits the known empirical facts of the actual universe at any given time. It is therefore useful to examine alternative universes and to study the differences in their formulation and empirical predictions, in order, for example, to devise new experiments for deciding among them. In systems theory, however, the situation is different. Since the criterion for acceptability is reliability, security, and related notions, there is little sense in studying systems theories that do not guarantee reliability and security, when there is already a clear and explicit theory that does.

In practice, furthermore, the software engineer or systems designer does not have to worry about the strictly mathematical axioms, for tree, for example, any more than the aeronautical engineer has to worry explicitly about differentiable manifolds when designing a new airplane. The general notions of tree and of what it means to decompose a function are intuitively clear, so we can concentrate our attention on the six specific HOS axioms in order to make sure that we decompose the functions correctly. Our trees will then be well-formed and reliability and security will be guaranteed.

In general, we can summarize the significance of the six HOS axioms in the following terms. Despite the high level of sophistication of contemporary systems analysis, the field has suffered from a serious defect. The system-specification process is itself a system, but, ironically, it has generally been carried out in an unsystematic fashion.

Much of what systems designers could learn from each other has often been lost in the shuffle; new systems have commonly had to be started from scratch. There has been no way to guarantee the efficiency of a system ahead of time. There have been problems of interface correctness, especially in complex systems designed by a large group of individuals, and subsystems can be included which are superfluous. No general means of guaranteeing system security has been available. Overspecification of a software system can detract from its transferability from one machine to another. The failure to separate specification clearly from implementation thus can unintentionally rule out the most efficient implementation of a given system.

Let us say that a system specification is functionally adequate, if it does what its designer wanted it to do, that is, if it does carry out the function it was supposed to. As far as we can tell, it seems that most systems in use today are functionally adequate, in this sense, at least to the extent that they have been tested and used. Otherwise, they would not be in use at all. Let us also say that a system specification is fully adequate, if it does what it is supposed to do in the most effective and efficient possible way. A fully adequate system would thus be both reliable and secure, for example. As noted in the last paragraph, though apparently functionally adequate, most software systems in use today most likely are not fully adequate. For all the reasons noted and others, although the jobs software systems are intended to do get done, they get done with a lot of waste, of time, money, and manpower.

The purpose of developing a standardized system-specification methodology is to eliminate this waste. Given some generally applicable principles governing the specification of systems, we can reduce the problem of

guaranteeing full adequacy to that of guaranteeing functional adequacy. With the correct set of principles on possible (allowed) system specifications, we can guarantee ahead of time that any system defined in accordance with those principles that does what it is supposed to do automatically does so in the most effective and efficient possible way. These principles then constitute the axioms of our systems theory or specification methodology.

We can get a clearer idea of what a systems methodology is by considering explicitly what it is not. A priori one might interpret the term "methodology" (or "theory") in either of two possible ways. The most ambitious form of methodology one might hope to develop would be a discovery procedure [Cho57] for system specifications. A discovery procedure would be a mechanical (algorithmic) procedure or set of procedures that would automatically produce, from a given set of requirements and specifications, a system that would produce those specifications from those requirements. Ideally, if we could manage to develop such a discovery procedure, we could eliminate systems analysts and designers altogether. The discovery procedure would automatically produce the appropriate system for any desired purpose. At our present level of knowledge, however, and probably in principle, such a notion of methodology is unrealizable. The most we can hope for at this time is a theory of constraints on system specifications. Such a theory severely limits the kinds of system specifications an analyst can design by insisting that the design satisfy the constraints specified in the theory as its axioms. If the system specification is functionally adequate, and if the designer has adhered strictly to the constraints provided by the theory, then the theory guarantees that it is fully adequate as well.

Developing such a theory of constraints places systems analysis on a par with the already developed natural sciences. When a physicist or chemist performs an experiment and observes a new phenomenon, for example, he tries to construct a theory that explains it. There is no discovery procedure that automatically produces a theory from the observations. The human scientist must use his ingenuity to construct the theory, just as the human systems analyst must use ingenuity in designing a system.

What the scientist does have available, however, is a theory of constraints on possible theories that limits the options available. Any theory the scientist proposes must guarantee conservation of mass-energy and of momentum, for example, and must be consistent with the laws of thermodynamics. These principles serve as axioms which any acceptable scientific theory must satisfy. What HOS provides for systems analysis, analogously, is a set of axioms (principles) which any fully adequate system specification must satisfy.

REFERENCES

Beh74 Behnke, H. et al. Fundamentals of Mathematics, Vol I: Foundations of Mathematics/The Real Number System and Algebra, translated by S. H. Gould, The MIT Press, Cambridge, MA, 1974.

Bir70 Birkhoff, G. and Lipson, J.D. "Heterogeneous Algebras," Journal of Combinatorial Theory 8, 1970.

Gut75 Guttag, J. "The Specification and Application to Programming of Abstract Data Types." Univ. of Toronto Technical Report CSRG-59, Sept, 1975.

Ham76a Hamilton, M. and Zeldin, S. "AXES Syntax Description," Technical Report #4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

Lin76 Linden, T. A. "Operating System Structure to Support Security and Reliable Software." ACM Computing Surveys, VIII, 4. Dec. 1976, pp. 409-445.

Wal75 Walter, W. C., et al. "Structured Specification of a Security Kernel." Proceedings, International Conference on Reliable Software, Los Angeles. April 21-23, 1975.

Cus77 Cushing, S. "The Software Security Problem and How to Solve It." Technical Report #6. Higher Order Software, Inc., Cambridge, MA, July 1977.

Ham76b Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software." IEEE Transactions, Vol. SE-2, No. 1, March 1976.

Ham76c Hamilton, M. and Zeldin, S. "Integrated Software Development System/Higher Order Software Conceptual Description." Version 1. Higher Order Software, Inc., Cambridge, MA, Nov. 1976.

Rya75 Ryan, M.P. and Shepley, L.C. Homogeneous Relativistic Cosmologies. Princeton University Press, Princeton, NJ, 1975.

Cho57 Chomsky, N. Syntactic Structures, Mouton, The Hague, 1957.

PART 4

Algebraic Specification of Data Types In Higher Order Software (HOS)

by
S. Cushing

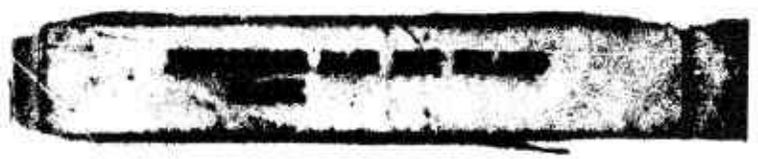
Algebraic Specification of Data Types In Higher Order Software (HOS)

1. INTRODUCTION: HOS AS A FORMAL SPECIFICATION METHODOLOGY

Higher Order Software(HOS) is a formal methodology developed by Margaret Hamilton and Saydean Zeldin [1] for the specification of large computer-based systems in a manner entirely independent of their implementation in particular configurations of hardware and resident software. Any such system can be formally specified in HOS in terms of three theoretical constructs: data types, functions (or operations), and control structures[1]. Data types are the kinds of objects a system operates on or produces; functions are the system components which operate on or produce members of data types; control structures are the relationships in accordance with which functions can be combined or decomposed. The purpose of this paper is to develop these notions from the point of view of data-type specification, which in HOS is done algebraically. In Section 2 we will examine the notion of function and develop the notions of algebra and control map, which play key roles in both data-type specification and system decomposition[2]. In Section 3 we will compare the theory developed in Section 2 with some other proposals for algebraic data-type specification that have appeared in the recent literature.

2. FUNCTIONS, ALGEBRAS, AND CONTROL MAPS

A function is a many-one correspondence between two sets. Members of the first set, called the domain, are said to be mapped by the function onto members of the second set, called the range. While a given range element may be mapped onto by any number of domain elements, it is crucial to the notion of a function that every domain element map onto exactly one range element. If x is a member of the domain of a function f, then the member y of the range of f that x maps onto is said to be f(x), the value of f at x, and we write "y = f(x)". The elements x of the domain are also said to be inputs of f, and the elements of the range are said to be possible outouts of x, since some range elements may not get mapped onto by f for any elements of the domain. The actual elements f(x) of the range that do get mapped onto by f for some x in the domain are said to be outputs of f.



In general there is no problem in specifying the domain and range of a function. We can simply state that the domain is some set and that the range is some set and, as long as these sets are clearly known to us, we are done. Specifying the actual mapping of a function is more problematic, however, especially in the case of infinite sets or sets which, though finite, are so large that they might just as well be infinite. For a small set, we can simply list a set of ordered pairs that contain only those values that the function actually makes correspond to each other. For a function defined on an infinite set, however, the corresponding set of ordered pairs would be infinite, so we could never explicitly specify the complete mapping in this way.

In some cases, we can get around this problem by giving an explicit rule or algorithm that expresses the mapping in general terms, without having to state the correspondence specifically for each domain element. In general, however, we may not always be able to specify such an algorithm directly; even if we can, furthermore, we may not want to specify a specific algorithm, because this might tie us in too tightly to a specific hardware or resident-software configuration. For such cases we introduce the notion of an algebra as a way of specifying indirectly what mapping our function is supposed to perform.

An algebra is a collection of sets and a collection of functions which map from and into those sets. Formally, we say that an algebra is an ordered pair $[\Sigma,\omega]$, where Σ is a collection of sets and ω is a collection of functions whose domains are Cartesian products of members of Σ and whose ranges are members of Σ[3]. The members of Σ are called the types[4] of the algebra, and the members of ω are called its primitive operations[5]. If Σ contains exactly one member, the algebra is said to be homogeneous, while if Σ contains more than one member, the algebra is said to be heterogeneous [2].

If the primitive operations of an algebra are simple enough functions, then we can characterize them just by listing their ordered pairs or by giving an explicit rule. If they are more complicated, however, then we can characterize them implicitly by giving axioms that describe their interactive behavior, without stating how to calculate their values for specific domain elements. An axiom, in this context, is a statement that asserts the equality of the outputs of two distinct combinations of functions and inputs. Given a sufficiently large and well-chosen collection of axioms, we can narrow down the class of functions that satisfy them to exactly those that we are trying to characterize.

Probably the simplest and most familiar class of algebras are the groups [3]. A non-empty set G is said to be a group with respect to a binary operation (function) Mult, called the group multiplication defined on G, if (1) G is closed under Mult, i.e., $Mult(g_1,g_2)$ is in G whenever g_1,g_2 are in G; (2) Mult is associative, i.e., the grouping of inputs is irrelevant so that $Mult(g_1,Mult(g_2,g_3))$ has the same value as $Mult(Mult(g_1,g_2),g_3)$; (3) there is an element in G that is neutral with respect to Mult, i.e., that always gets mapped with some other input by Mult onto that other input; and (4) every element of G has an inverse element, i.e., an element which gets mapped with it by Mult onto the neutral element.

For example, the positive rational numbers form a group if we take Mult to be Multiplication, but the positive integers do not, since there are then no inverse elements. The non-zero rational numbers form a group, if we take Mult to be multiplication and all the rationals form a group if we take Mult to be addition. The integers form a group if we take Mult to be addition, but the integers do not form a group if we take Mult to be multiplication, because, again, there are then no inverse elements.

We can specify the groups formally as homogeneous algebras by giving an algebraic specification.

consisting of three parts. First we give a type specification which tells us that Σ contains a single set G which contains a distinguished element Neut:

$$\Sigma: \quad G, \text{Neut} \in G$$

Second, we give an operation specification which tells us that ω contains two elements: Mult, whose domain is G x G and whose range is G, and Inv, whose domain and range are each G.

$$\omega: \quad \text{Mult}: \ G \times G \to G$$
$$\text{Inv}: \ G \to G$$

Third, we give an axiom specification which states formally that Mult and the members of G behave in the way we said earlier they should:

Axioms: For all $g, g_1, g_2, g_3 \in G$,

1. $\text{Mult}(g_1, \text{Mult}(g_2, g_3)) = \text{Mult}(\text{Mult}(g_1, g_2), g_3)$
2. $\text{Mult}(\text{Neut}, g) = g$
3. $\text{Mult}(g, \text{Neut}) = g$
4. $\text{Mult}(g, \text{Inv}(g)) = \text{Neut}$

Axiom 1 says that Mult is associative, Axioms 2 and 3 say that the distinguished element Neut is indeed a neutral element, and Axiom 4 says that Inv does, in fact, produce inverse elements with respect to that neutral element. The remaining property that we gave for groups, i.e., that G is closed under Mult, is guaranteed by the operation specification, because of the definition of "function", so we do not have to include it in the axioms.

The algebraic specification we have just given is the specification not of a group, but of the groups as a class. A group consists of a particular set on which functions with the appropriate behavior have been defined. Our specification leaves the set G uncharacterized, however, telling us that any set G whatsoever can qualify as a group, as long as there are operations definable on that set which satisfy our operation specification and our four axioms.

Algebraic specifications are used in HOS to characterize all sorts of objects other than purely mathematical structures like groups. We can characterize the notion of a stack, for example, a data type which has been extensively discussed in the literature (e.g., [4],[5],[6],[7]), in terms of three primitive operations and four axioms, as illustrated in Figure 1.

```
DATA TYPE:  STACK;
PRIMITIVE OPERATIONS:
  stack1 = Push(stack2,Integer1);
  stack1 = Pop(stack2);
  Integer1 = Top(stack1);
AXIOMS:
  WHERE Newstack IS A CONSTANT STACK;
  WHERE s IS A STACK;
  WHERE I IS AN INTEGER;
Top(Newstack) = REJECT;
Top(Push(s,I)) = I;
Pop(Newstack) = REJECT;
Pop(Push(s,I)) = s;
END STACK;
```

Figure 1

Data Type STACK

The algebraic specification in Figure 1 is written in the HOS specification language AXES [8] and characterizes the notion stack of integers. The operation specification, which gives the domains and ranges of the members of ω, appears under the heading "PRIMITIVE OPERATIONS", and the type specification which gives the members of Σ, along with any distinguished elements, is provided by the WHERE statements. The type specification tells us that there are two sets in this algebra, the set of stacks and the set of integers, so the algebra is heterogeneous, and that there is a distinguished stack called Newstack. The set of integers is assumed to have been characterized independently in terms of its own algebraic specification [9].

The operation specification tells us that there are three primitive operations in this algebra, Push, Pop, and Top. Intuitively, Push is the operation that places an element on a stack, Pop is the operation that removes an element from the top of a

stack and discards it, and Top is the operation that tells us what the top element of a stack is, without removing it from the stack. These intuitive characterizations are specified formally in the axioms.

The axiom specification appears immediately after the WHERE statements. The second and fourth axioms characterize a stack as a last-in/first-out storage device. The second axiom says that pushing an integer onto a stack produces a stack whose top element is that integer. The fourth axiom says that popping a stack onto which an integer has been pushed produces the stack it was pushed onto. The first and third axioms characterize the distinguished element Newstack as the stack that contains no elements, since it has no top element and cannot be popped. REJECT is an ideal element, like $\sqrt{-1}$ and the empty set [3]; it is assumed to be a member of every type, but to be invisible to universal quantification[6]. An operation is said to have an output of REJECT when it has no genuine output of the expected sort.

Another example, that of time, is shown in Figure 2. This specification defines data-type TIME in terms of three primitive operations. Advance is the operation of beginning at the time indicated by the first argument and advancing by the amount of time indicated by the second argument. Notafter is the relation, i.e., boolean-valued function, that holds between two times if the first is earlier or simultaneous with the second. Reverse is the operation that maps each time onto its mirror image with respect to the null element Notime. Axioms 1-3 characterize Notafter as a partial ordering and Axiom 4 makes that ordering total. Axiom 5 characterizes Notime as the neutral element with respect to Advance. Axiom 6 says that Advance is commutative and Axiom 7 says that it is associative. Axiom 8 says that Advance moves up in the partial ordering if the amount it advances by is above Notime in the partial ordering. Axiom 9 says that Reverse produces the reverse of a time with respect to Notime.

Using Notime, rather than something like Precedes, simplifies the axioms somewhat, while omitting Reverse would result in a slightly different data type and thus a different notion of time. On the one hand, more implementations would then be possible, since the specification would not require all of them to support the possibility of reversal; on the other hand, fewer features of each allowed implementation could be made use of, since the specification would not permit the use of reversal even in those implementations that could support it. As usual, which notion of time we choose depends entirely on the system we are dealing with and what we want it to be capable of.

DATA TYPE: TIME;
PRIMITIVE OPERATIONS:

$time_3$ = Advance($time_1$,$time_2$);
boolean = Notafter($time_1$,$time_2$);
$time_2$ = Reverse($time_1$);

AXIOMS:

WHERE t,t_1,t_2,t_3 ARE TIMES;
WHERE Notime IS A CONSTANT TIME;

1. Notafter(t,t) = True;
2. Entails(Notafter(t_1,t_2) & Notafter(t_2,t_3), Notafter(t_1,t_3)) = True;
3. Entails(Notafter(t_1,t_2) & Notafter(t_2,t_1), Equal(t_1,t_2)) = True;
4. Notafter(t_1,t_2) | Notafter(t_2,t_1) = True;
5. Advance(t,Notime) = t;
6. Advance(t_1,t_2) = Advance(t_2,t_1);
7. Advance(t_1,Advance(t_2,t_3)) = Advance(Advance(t_1,t_2),t_3);
8. Notafter(Advance(t_1,t_2),t_1) = Notafter(t_2,Notime);
9. Advance(Reverse(t),t) = Notime;

END TIME;

Figure 2
Data Type TIME

The operations Entails, &, and Equal, which appear in Axioms 2 and 3, are assumed to have been characterized independently, the first two on type BOOLEAN [9] and the third as a universal operation on any type.

In our description of the algebraic specification in Figure 1, we noted that the set of integers is assumed to have been characterized independently in terms of an algebraic specification of its own. In general, in the characterization of a particular type, the relevant algebra will contain a number of sets (in Σ), all but one of which has been characterized already and so can serve as a basis for defining the new type of interest. Mathematically, we could just as well view the various sets as mutually characterizing each other via the relationships expressed in the axioms, just as we view the type of interest and the primitive operations as mutually characterizing each other, but this point of view can get rather confusing.

Once we have our object types algebraically characterized, we can then go ahead and define more complicated operations, either on a single type or on more than one type, in terms of the primitive operations on the types. Given our algebraic specification of groups, for example, we can define a power function on groups, as illustrated in Figure 3. The tree structure in Figure 3, called a control map, gives a complete specification of how to find the n^{th} power of any group element. It could be simplified considerably through the use of various abbreviatory notations and conventions, but it has been intentionally written out here in full in order to illustrate as many of the available formal devices as possible.

The functions K_i, $Clone_i$, and $Identify_j^i$ are universal operations defined on every type. K_i is the constant function, which produces i as output for every input. $Clone_i$ is the clone function, which produces as output i copies of whatever it takes as input, in effect producing new names for its input.

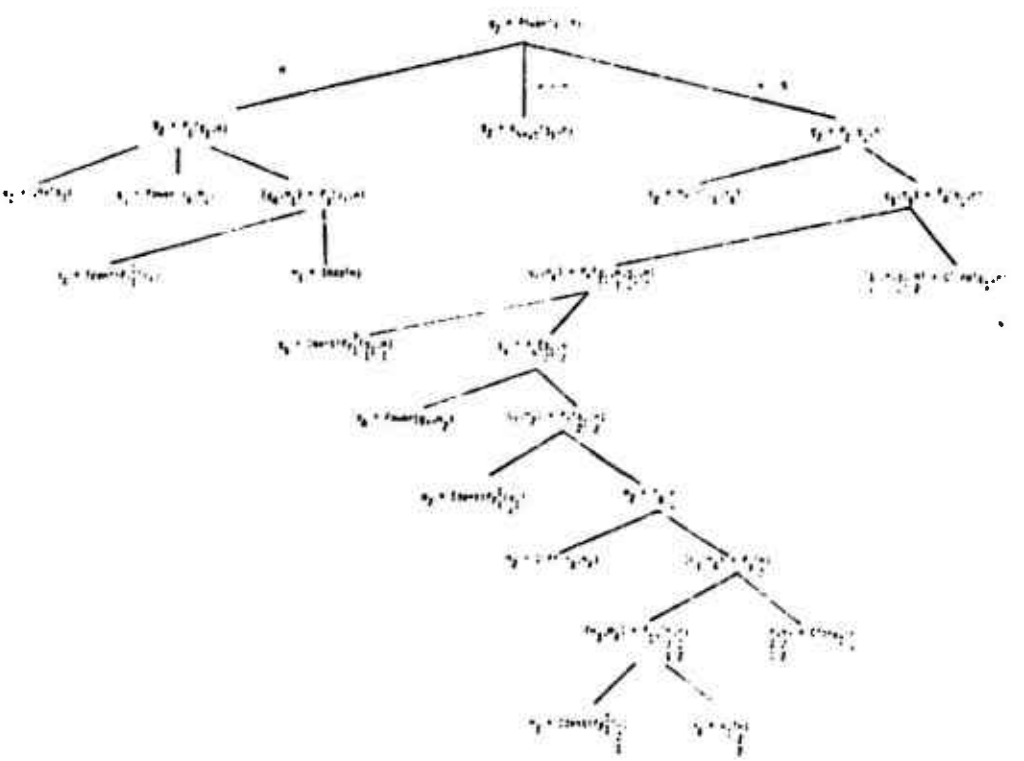


Figure 3
Control Map for the Power Function on Groups

$Identify^i_j$ is the identify function that takes i-tuples as input and produces as output the j^{th} components of those tuples. K_i and $Identify^i_j$ are standard functions in mathematics and $Clone_i$ is very useful as a way of guaranteeing that every variable in a software system gets used as input and output exactly once. This aids in keeping track of variables so the efficiency of storage use can be maximized.

The symbols "f_i" in Figure 3 are simply names assigned to the functions performed by tree subconfigurations and have no further significance. Every non-terminal node except the topmost is occupied by such a function, whose inputs and outputs are as indicated by the typed variables shown and whose mapping is indicated by its decomposition into subfunctions. Every terminal node, except those occupied by the function being defined, is occupied by either a primitive operation on one of our types, a universal operation, or an operation that is assumed to have been already characterized by its own control map on one (or more) of our types. Diff, meaning difference, and Iopp, meaning integer opposite (output = -n for input = n), for example, might be primitive operations on the integer type or control map functions, depending on what our algehraic specification of the integers looks like [9]. The topmost node, of course, is occupied by the function being defined, whose mapping is characterized by the entire tree. This function can also appear at terminal nodes, indicating recursive evaluation, but its variables must must be different at each appearance in order to avoid circularity.

The real meat of a control map is in its control structures, which determine how a function relates to those into which it is decomposed. The tree in Figure 3 exemplifies three primitive control structures: set partition, class partition, and composition. The set partition primitive control structure is illustrated by the immediate decomposition of the topmost function. It partitions the domain of the function into non-overlapping, exhaustive sets and specifies the restriction of the decomposed function to each of those sets. In Figure 3, the integers are partitioned into those which are less than, equal to, and greater than zero, and the restrictions of the Power function to those sets are specified accordingly. Functions f_3, f_5, f_7 and f_{10} illustrate the class partition primitive control structure, which matches specific inputs and outputs in an exhaustive and non-overlapping way. The composition primitive control structure, which makes the output of one subfunction the input of another, is illustrated by functions f_1, f_2, f_4, f_6, f_8, and f_9. Note that, while decomposition in a control map need not be binary, as illustrated by f_1 and the topmost node, it can always be made binary by the introduction of new f_i functions with the appropriate control structures.

Mathematically, the effect of control maps seems to be to write arbitrary functions as polynominal functions on heterogeneous algebras except that recursion is allowed, as in Figure 3. Given an arbitrary function to be performed by some software system, for example, we may know what sets the function maps from and into, but we may not know what types it maps from and into. That is, we may not know what primitive operations will be required at the terminal nodes of our control map. The principles for constructing control maps facilitate the determination of these primitive operations and thus of the types involved in the software system our function is a part of.

Control maps can be simplified considerably by defining abstract control structures, which abbreviate recurrent combinations of the primitive control structures. For example, Figure 4a contains the control map for a Regress function on our data type TIME, defined entirely in terms of primitive control structures (in this case, one instance each of composition and class partition). The part of the figure that is surrounded by a dotted line, however, is an instance of a very common control structure, called COJOIN [10], which enables us to have the same variable accessed by more than one

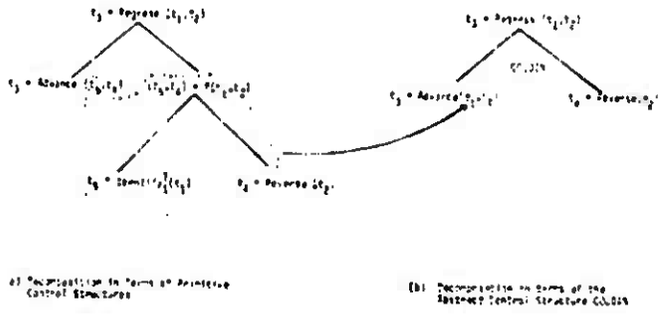


Figure 4

Two Control Maps for
Operation Regress on Data Type TIME

function in a system. In this case, the effect of the encircled tree subconfiguration is simply to guarantee that t_5 has the same value as t_1, as shown by the arrow. We can thus simplify the control map by replacing t_5 by t_1 and the relevant subconfiguration by the abstract control structure COJDIN, as shown in Figure 4b. The primitive control structures are needed for theoretical purposes, to guarantee system reliability, for example, as discussed in [1], but for convenience in practical system design abstract control structures may be used instead, as long as they are explicitly decomposable into the primitives. New abstract control structures of any complexity can be defined in a similar way, as discussed in [8] and [10].

One of the useful things about control maps is that they enable us to refine the notion of primitive operation in a very precise way. Given an algebra, with its associated primitive operations, we can say that some other function on the types of that algebra is <u>not</u> primitive, if it can be decomposed into the primitives by means of a control map. Any desired function on the types of the algebra that cannot be defined in te· ·f the primitives with a control map must then ·dded to the operation specification of the algebra as a new primitive, thereby creating a new algebra. Given an algebra, in other words, the control maps partition the set of functions definable on the types of the algebra into two classes: those which are representable as control-map functions on that algebra and those which are not so expressable. New algebras can be created by adding members of the latter class successively as primitive operations to the operation specification of the original algebra, with a new algebra being created with every addition.

Functions which are representable as control maps need not always be represented as control maps, however. AXES allows us to write such functions as what are called derived operations; these are characterized by providing assertions that specify the interactive behavior of the function with other functions that have been characterized independently. Given an algebraic specification of the natural numbers as a data type with the successor function as one of its primitive operations and a boolean-valued control map-defined function Factor that maps a pair of naturals onto True if the first is a factor of the second and onto False otherwise [9], we can define a greatest-common-denominator function GCD in AXES as follows:

DERIVED OPERATION: n_3 = GCD(n_1,n_2);

WHERE n_1,n_2,n_3,n_4 ARE NATURALS;

Factor(GCD(n_1,n_2),n_1) = True;

Factor(GCD(n_1,n_2),n_2) = True;

Entails(And(And(Factor(n_4,n_1),Factor(n_4,n_2)),
Not(?Equal?(n_4,Zero))), Factor(n_4,
GCD(n_1,n_2))) = True;
END GCD;

What this definition says is that the GCD of two natural numbers is a natural number which is a factor of both and which has every factor of both as a factor of its own. The function is thus defined in terms of its behavior with respect to other functions, rather than in terms of its decomposability into other functions. Similarly, we could just as well have defined our Regress function on data type TIME as a derived operation as follows:

DERIVED OPERATION: $time_3$ = Regress($time_1$, $time_2$);
WHERE t_1,t_2 ARE TIMES;
Advance(Regress(t_1,t_2),t_2) = t_1;
END Regress;

There is no difference between the functions, i.e., mathematical mappings, that we get from control-map definitions and those that we get from derived-operation definitions. The only difference is in how we choose to specify the functions, and this is largely a matter of convenience.

The assertions used in defining derived operations look very much like the axioms that are used in defining primitive operations, but their status is really quite different. Given the set of primitive operations for some data type, we can prove mathematically the existence of other functions; it is these functions, such as GCD (see, e.g., [11] for a simple existence proof), that may be specified as control-map operations or derived operations. Explicitly specifying these latter functions does not change the algebra underlying the data type in any way, since their existence as mathematical objects is already entailed by the algebra. Sometimes, as in the case of GCD in fact, the proof may even provide guidelines to the construction of an appropriate control map, but it is up to us whether we actually go ahead and construct the control map or whether we treat the function as a derived operation. Functions whose existence is not so entailed, however, must be added to the primitive operation specification and characterized in terms of axioms, thus creating a new algebra, as noted earlier.

3. DISTINCTIVE FEATURES OF HOS

The theory we have developed here differs from other proposals for algebraic data-type specification that have appeared in the literature ([4],[6], [7],[12],[13]) in at least three very important ways[7]. First, HOS distinguishes very sharply, in concept, between the specification of a software system and its implementation. Liskov and Berzins [12] require that "All objects of an abstract data type must have been produced by some sequence of the constructor operations of that type" (p. 13-12), and Guttag [4] states that "the need for operations to generate values of the type is clear" (p. 45). In our opinion, this requirement reflects a confusion between specification and implementation. Clearly, an implemented system must have some way of generating the objects it deals with. It does not follow, however, that the manner of generation must necessarily be included in the specification of the system. For a particular system we are interested in, it simply may not be important for us to know how the objects got to be the way they are. As Liskov and Zilles [7] point out,

> It should be possible using the specification method to construct specifications which describe the interesting properties of the concept and nothing more. The properties which are of interest must be described precisely and unambiguously but in a way which adds as little extraneous information as possible. In particular, a specification must say what function(s) a program should perform, but little, if anything, about how the function is performed. One reason this criterion is desirable is because it minimizes correctness proofs by reducing the number of properties to be proved (p. 9).

It seems clear, in these terms, that the manner of generation of objects may very well be a part of the "what" of a system, but that it could equally well be a part of the "how". Some systems will include object-generating functions in their specifications and some will not. We cannot stipulate

ahead of time which of these might be the case for any system we might come across[8].

A second difference between HOS and other theories is its capacity to define non-primitive operations in a natural way, either as control-map operations or as derived operations. Control maps are unique to HOS and should require no further comment. A little more should be said, however, about derived operations, so that their usefulness can be fully appreciated.

We have noted that the assertions used in characterizing a derived operation look very much like axioms, but differ in that the existence of the operations they characterize can be proven mathematically from axioms we already have. The very fact that we must construct such a proof, however, each time we introduce a derived operation significantly simplifies the proof of consistency of the axioms themselves. Guttag, Horowitz, and Musser [6], for example, apparently allow us to define operations only by including axioms that characterize them in the axiom specification of an algebra. In a large system, therefore, the number of "axioms", and thus the task of proving their consistency, could turn out to be considerable. In HOS, in contrast, we must test each new non-control map operation to see if the existence of its functions follows from the axioms we already have; if so, we need add no new axioms to our algebra, because the operation is not primitive, but derived. In effect, the use of derived operations enables us to modularize the consistency proof for our axioms. Whereas Guttag, Horowitz, and Musser might end up with the formidable task of having to prove the consistency of a hundred "axioms", we might be able to decompose that task into several smaller proofs, each involving a proper subset of the proposed "axioms", with one such subset for each derived operation. The savings in time and effort could be considerable, and the proofs could also provide insights into constructing appropriate control maps, should we decide later that this is desirable.

A third distinguishing feature of HOS is its natural suitability for the specification of large and very large software systems. After reviewing several specification techniques for data abstraction, including a version of algebraic data-type specification, Liskov and Zilles [7] point out that

> The specification techniques discussed in this paper can adequately describe modules--the blocks of which systems are built--but it is not clear that they can describe the entire system. For example, Parnas has shown how a KWIC system can be modularized... and each module was described using his specifications, but the specification of the system as a whole was given in English. It seems unlikely that an entire system can be viewed as a single, top-level module, so perhaps a different kind of specification technique is desirable here (p. 18).

In our opinion, the inadequacy that Liskov and Zilles point out results from a confusion between the need to decompose a system into modules and the need to characterize the kinds of objects the system deals with. In HOS the latter need is satisfied by algebraic data-type specifications and the former by the use of control maps. Once the data types of a system are formally characterized, the overall function the system performs can be decomposed into subfunctions arranged hierarchically in a control map. It is these subfunctions, along with the data types that serve as their domains and ranges, and not the data types themselves, that constitute the modules into which the system is decomposed. It is the control map that serves as the formal specification of the "entire system".

As Hamilton and Zeldin suggest [14], there is good reason to believe

> that the basic properties of "large" systems are not really different than those of "small" systems. It is only that small systems are kinder, yet more deceptive, in not displaying their real properties. But the time has come when one is forced by large systems to look at properties of systems. They are more basic than one cares to admit (p. 1).

From a mathematical point of view, for example, the notion of control map seems to follow quite naturally from that of algebraic data-type specification, by combining the notion of polynominal

function on a heterogeneous algebra with that of recursive call; it was only by careful empirical examination of the properties of large real-world systems, however, that the notion of control map was actually developed and its relationship to algebraic data-type specification discovered. Lisko[illegible]d Zilles' observation about the inadequacy of other approaches to algebraic data-type specification underscores the need for careful analysis of the actual properties of large systems in developing a general specification methodology.

FOOTNOTES

1. Hamilton and Zeldin also talk about the layers of a system, but we will not deal with these here. See [15], [16] for some discussion of this important notion.

2. In an otherwise very good review of HOS [17], Peters and Tripp make the erroneous statement that in HOS "the issue of data base design was, at best, addressed implicitly" (p. 94). Actually, as we will see in this paper, some of the better-known aspects of HOS, such as control maps, follow quite naturally from its manner of specifying data types and vice versa. Algebraic data-type specification and control-map function specification are complementary aspects of a complete systems theory and neither makes much sense, in our opinion, without the other. We thus fully agree with Peters and Tripp that "the design of code and data base must be synchronous". See [8] and [18] for further discussion.

3. In HDS we also allow the ranges to be Cartesian products of Σ members to maximize the generality of the theory. No mathematical problems arise from this extension, as far as I can tell.

4. In [9] I called these categories because of their similarity to the syntactic, semantic, phonological, and lexical categories of linguistics [19], but I think this term might be misleading, since "category" has a somewhat different meaning in mathematics [20]. The original term, "phyla", used by Birkhoff and Lipson when they first introduced heteroganeous algebras [2] I find somewhat unnatural, but this is, of course, just a matter of taste. The word "type", in this context, is due to Guttag [4].

5. The AXES specification language uses the terms "function" and "operation" in slightly different ways [8], but this difference is not relavant for us here, so we will use the two terms interchangeably.

6. Guttag, Horowitz, and Musser [6] object to the use of elements like REJECT which "are implicitly included in all types" (p.65) because, they argue, "then one can no longer always assume that the axioms are universally quantified over the types". This objection presupposes, however, that the meaning of "universal quantification" is somehow known a priori, rather than having to be specified explicitly in the formal semantics of our specification language. Given an explicit formal semantic theory, it is a simple matter to include in the semantic rule for "universal quantification" a statement to the affect that REJECT is ignored in the assignment of truth values. See [21], [22], and [19] for extensive discussion of the form such a theory might take.

7. See, also, footnote 6.

8. Guttag's decision [4] to treat an individual object "as a nullary 'operation', rather than as a 'value'" (p. 43) reflects a particular instance of this conceptual confusion, we think. Nullary operations are degenerate objects, like point circles and circular ellipses, which, though counterintuitive, are often introduced in mathematics to maximize elegance and simplify proofs, in part through the elimination of exceptional cases. (See [9] for some discussion of the counterintuitive character of nullary operations.) In Guttag's system, however, they seem to play no role other than that of identifying distinguished objects, such as Zero and Newstack, so it seems just as well to dispense with such operations altogether. In HOS we specify the existence of distinguished objects by simply stating that the objects exist; this is done in the WHERE statements of a data-type specification, as we saw in Figures 1 and 2. We thus state what distinguished objects a system deals with without saying how they got to be the way they are in particular implementations.

REFERENCES

[1] Hamilton, M. and S. Zeldin, "Higher Order Software--A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976, pp. 9-32.

[2] Birkhoff, G. and J.D. Lipson, "Heterogeneous Algebras", Journal of Combinatorial Theory, 8, 1970, pp. 115-133.

[3] Bahnke, H., F. Backman, K. Fladt, and W. Süss (eds.), Fundamentals of Mathematics, Volume 1: Foundations of Mathematics/The Real Number System and Algebra, MIT Press, Cambridge, MA, 1974.

[4] Guttag, J.V., "The Specification and Application to Programming of Abstract Data Types", Technical Report CSRG-59, University of Toronto, Toronto, September 1975.

[5] Parnas, D.L., "A Technique for Software Module Specification with Examples", Comm. of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336.

[6] Guttag, J.V., E. Horowitz, and D.R. Musser, "Some Extensions of Algebraic Specifications", in Wortman, D.B. (ed.), Proc. of an ACM Conference on

Language Design for Reliable Software, Raleigh, NC, March 28-30, 1977, Association for Computing Machinery, New York.

[7] Liskov, B.H. and S.N. Zilles, "Specification Techniques for Data Abstractions", IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 7-19.

[8] Hamilton, M. and S. Zeldin, "AXES Syntax Description", TR-4, Higher Order Software, Inc. (hereafter cited as HOS, Inc.) Cambridge, MA, Dec. 1976.

[9] Cushing, S., "Algebraic Specification of Abstract Data Types/The Intrinsic Types of AXES", Appendices III and IV of [8].

[10] "Techniques for Operating System Machines", TR-7, HOS, Inc., Cambridge, MA, July 1977.

[11] Lange, S. Algebraic Structures, Addison-Wesley, Reading, MA, 1967.

[12] Liskov, B.H. and V. Berzins, "An Appraisal of Program Specifications", to appear in [23].

[13] Zilles, S., "Algebraic Specification of Data Types", Progress Report 11, Laboratory for Computer Science, MIT, Cambridge, MA, 1975.

[14] Hamilton, M. and S. Zeldin, "Discussion of an Appraisal of Program Specifications", to appear in [23].

[15] Hamilton, M. and S. Zeldin, "Integrated Software Development System/Higher Order Software Conceptual Description", TR-3, HOS, Inc., Cambridge, MA, Nov. 1976.

[16] Hamilton, M. and S. Zeldin, "The Manager as an Abstract Systems Engineer", Digest of Papers, Fall COMPCON 77 (Washington, D.C.), IEEE Computer Society, Cat. No. 77CH1258-3C, Sept. 6-9, 1977.

[17] Peters, L.J. and L.L. Tripp, "Comparing Software Design Methodologies", Datamation, Nov. 1977, pp. 89-94.

[18] Cushing, S., "The Software Security Problem and How to Solve It", Revision 1, TR-6, HOS, Inc., Cambridge, MA, July 1977, reprinted in [10].

[19] Cushing, S., "An Algebraic Approach to Lexical Meaning", presented at the Annual Meeting, Linguistic Society of America, Chicago, Dec. 28-30, 1977.

[20] MacLane, S., Categories for the Working Mathematician, Springer-Verlag, New York, 1971.

[21] Cushing, S., "The Formal Semantics of Quantification", UCLA dissertation, available from the Indiana University Linguistics Club, Bloomington, Indiana.

[22] Cushing, S. "The Formal Representation of Truth Conditions in the Lexicon of a Grammar", presented at the Interdisciplinary Linguistics Conference on Approaches to the Lexicon, University of Louisville, Louisville, Kentucky, March 10-12, 1977.

[23] Wegner, P., J. Dennis, M. Hammer, and D. Teichroew, Proc. of the Conference on Research Directions in Software Technology, to be published by MIT Press in Spring 1978.

PART 5

Software Engineering,
Artifical Intelligence and Cognitive Processes

by

S. Cushing

Software Engineering, Artificial Intelligence, and Cognitive Processes

Work in language and mind in artificial intelligence (AI) presupposes a parallelism between minds and information-processing systems, a parallelism that is also assumed or argued for by some non-AI researchers (Miller, 1976;Miller and Johnson-Laird,1976;Fodor,1975). We agree that such a methphor can be useful in gaining insight into the nature of cognitive processes, but we think that this is possible only if a number of conditions are satisfied. First, we think that any adequate computer model of the mind will have to be formulated not in terms of programs, as in current AI work, but in terms of software systems, as discussed in the newly emerging field of software engineering.[1] We find it highly implausible that complex mental processes can be modeled adequately in terms of sequential lists of instructions, which programs, by definition, are. The mind is a highly complex system of related and interacting, but essentially autonomous components,[2] and it seems likely that some of the more interesting generalizations concerning its structure and operation will involve the interfaces between these components, at least as much as the individual programs that may make them up. Not surprisingly, it is precisely in regard to their interfaces that some of the more interesting properties of software systems have emerged (Hamilton and Zeldin, 1976a,b).

Second, we think that such an approach to the study of mind would require a genuine theory of software systems, rather than the sort of ad hoc programming that is endemic to current AI work. Dresher and Hornstein (1976, 1977, 1978) argue that natural language-related work in AI has been generally devoid of explanatory value because of the ad hoc character of the programs involved. Most of this work seems to be not scientifically, but technologically oriented, i.e., geared toward developing machines that can process sentences of natural language, rather than seeking general principles that can serve as genuine scientific explanations of linguistic phenomena. The ad hoc character of computer programming has become a serious problem much more generally, however, particularly in connection

with the specification of large and very large software systems, and it is precisely this problem that motivated the development of software engineering in the first place.[3] While this motivation has also been primarily technological, e.g., minimizing cost in the development of large systems, its aim is to develop a general theory of software systems that accounts for their essential properties in a principled way.[4] Such a theory might very well be of genuine scientific interest, precisely because of its concern for general explanatory principles.

Such a theory would characterize the notion "possible software system" and, in accordance with the parallelism mentioned above, could thus be taken equally as characterizing the notion "possible mind," just as the notion "possible grammar" as a device that generates structured strings of objects is viewed in linguistics as providing an abstract characterization of the notion "possible language." Given such a formal characterization of "possible mind," we might then be able to constrain it in accordance with known empirical facts to get a notion of "possible X mind," where X = "human" or any other species, just as linguists try to constrain grammars to get a characterization of "possible human language." The notion "possible human mind" might then be further constrainable in accordance with the idiosyncratic facts of an individual's culture and life experience, giving us an explanatory account of an individual human mind and its associated behavior. The difference between this approach and the one current in AI would be essentially the same as that between generative and taxonomic linguistics. Rather than starting from scratch, as it were, and building up programming systems ad hoc, we would be beginning with a principled account of the sort of entity we assume the human mind to be and then narrowing that account down in accordance with empirical facts to determine precisely which entities of that sort the mind really is.

Hamilton and Zeldin (1976a,b,c,d) argue that the notion "possible software system" can be formalized in terms of three theoretical constructs--data types, the kinds of entities that systems operate on or produce; functions (or operations), the entities that operate on or produce members of data types; and control structures, the relationships in accordance with which functions can be decomposed or combined--and that

each of these constructs can exist on various layers, which are strikingly reminiscent, in concept, to the "levels of description" of generative grammar. They also provide a formal methodology for representing these constructs abstractly, in terms that are entirely independent of a system's implementation in particular configurations of hardware or resident software (operating systems, etc.). Their theory seems to imply that a software system can be characterized as a set of homomorphically related polynomial functions on heterogeneous algebras (Cushing, to appear).[5] To the extent that it does, in fact, capture the notion "possible software system," we can presumably take the theory as equally a characterization of "possible mind," and proceed to constrain it accordingly.

Cushing (1977a,c) argues that the notion of algebraic data-type specification that is incorporated in Hamilton and Zeldin's theory[6] provides a revealing model for the semantic lexicon of a natural language, as one component of the human mind. The model incorporates an empirical claim as to where in the lexicon we would most naturally expect to find constraints, as part of a general characterization of the kind of subcomponents that make it up. Cushing argues that the semantic lexicon is a heterogeneous algebra[7] and that some seemingly unrelated issues that have received attention in the recent linguistic and psycholinguistic literature (e.g., lexical decomposition vs. meaning postulates, functional notation vs. semantic markers) receive a natural and revealing reformulation, when viewed in this light.

We will not speculate here on how fruitful this line of research will ultimately turn out to be, because that can be determined only by time and further work. We do think, however, that something along these lines is a necessary prerequisite to a computer-based model of cognition. It may, in fact, turn out that the mind is not a computational device at all and that entirely new concepts will have to be developed to account adequately for its operation. Our point is simply that any adequate theory of mind will have to base itself firmly on the search for general explanatory principles and that this applies to computationally-based theories as much as to any other.

Footnotes

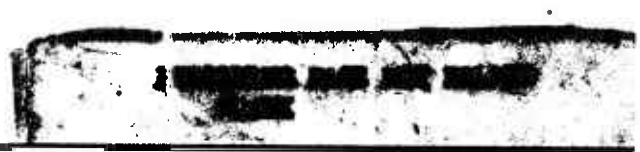
1. It is not surprising that the emergence of software engineering has yet to have an impact on AI. The field is so new that the first journal devoted specifically to it (IEEE Transactions on Software Engineering) did not appear until 1975, and, even as late as 1972, D.L. Parnas could still refer legitimately to "the so-called 'software-engineering' problem" (p. 330). For a general survey of some current trends in the field, see Wegner, Dennis, Hammer, and Teichrow (1978). (The fact that some authors still use the terms "program" and "system" interchangeably should not be allowed to obscure the important conceptual distinction involved. This difference should become clearer in what follows.)

2. Some recent physiological evidence which, it seems to us, can be naturally interpreted as directly supporting this contention is reported in Hochstein and Shapley (1976a,b), Levick (1975), and Werblin (1974). We thank Dr. Michael Zeldin of the Harvard Biology Laboratory for bringing this work to our attention.

3. See, for example, Asch, Kelliher, Locher, and Connors (1975), Corelli and Williams (1976), Hamilton (1971, 1972), Hamilton and Zeldin (1976c, Chapter 1; 1977), Ramamorthy and Ho (1975), and Richter and Mason (1976) for some discussion of this motivation.

4. See, for example, Bratman and Court (1975), Davis and Vick (1976), Hamilton and Zeldin (1973a,b;1976a,b,c,d; 1977), HOS (1977), Mills and Wilson (1976), Richter and Mason (1976), Robinson, Levitt, Neumann, and Saxena (1975), Robinson and Levitt (1977), Stevens, Meyers, and Constantine (1974), and Wilson (1976) for some specific proposals in this regard. For comparative discussion of a number of such proposals, see Cushing (1977b), Hamilton and Zeldin (1976c, Chapter 2), and Peters and Tripp (1977).

5. See Birkhoff and Lipson (1970) for a general discussion of heterogeneous algebras.



6. See Cushing (1976a) for elaboration of this notion and Guttag (1975) and Zilles (1975) for a very different approach to algebraic data-type specification. Some of the differences are discussed briefly in Hamilton and Zeldin (1978).

7. See Cushing (1976b,part IV;1976c;1977d) for some related ideas concerning the algebraic representation of lexical meaning.

Bibliography

1. Asch, A., D.W. Kelliher, J.P. Locher III, and T. Connors (1975) "DoD Weapons Systems Software Acquisition and Management Study Volume 1, MITRE Findings and Recommendations", Vol.1, MTR-6908, The MITRE Corporation, Bedford, Massachusetts, May 1975.
2. Birkhoff, G. and J.D. Lipson (1970) "Heterogeneous Algebras", Journal of Combinatory Theory, 8, 115-133.
3. Bratman, Harvey and Terry Court (1975) "The Software Factory", Computer, 8, 7.
4. Corelli, Robert R. and Thomas G. Williams (1976) "Management Overlay for the BMDATC Software Development System", System Development Corporation, Huntsville, Alabama, 15 December 1976.
5. Cushing, Steven (1976a) "Algebraic Specification of Abstract Data Types/The Intrinsic Types of AXES", Appendices III and IV of Hamilton and Zeldin (1976d).
6. Cushing, Steven (1976b) "The Formal Semantics of Quantification", UCLA dissertation, available from Indiana University Linguistics Club, Bloomington, Indiana.
7. Cushing, Steven (1976c) "The Group Structure of Quantification", UCLA Papers in Syntax, 7,1-7.
8. Cushing, Steven (1977a) "Lexical Decomposition and Lexical Algebra", presented to the MIT Workshop on Language and Cognition, February 1977.
9. Cushing, Steven (1977b) "The Software Security Problem and How to Solve It," Revision 1, TR-6, Higher Order Software, Inc. (hereafter cited as HOS, Inc.), Cambridge, Massachusetts, July 1977, reprinted in HOS (1977).
10. Cushing, Steven (1977c) "An Algebraic Approach to Lexical Meaning", to be presented at the Annual Meeting, Linguistic Society of America, Chicago, Illinois, December 1977.
11. Cushing, Steven (1977d) "The Formal Representation of Truth Conditions in the Lexicon of a Grammar", presented at the Interdisciplinary Linguistics Conference on Approaches to the Lexicon, University of Louisville, Louisville, Kentucky. March 1977.

12. Cushing, Steven (to appear) (work in progress).

13. Davis, C.G. and C.R. Vick (1976) "The Software Development System", in supplement to Proceedings of the 2nd International Conference on Software Engineering, San Francisco, IEEE Catalog No. 76h1125-4 C, October 1976

14. Dresher, B. Elan and Norbert Hornstein (1976) "On Some Supposed Contributions of Artificial Intelligence to the Scientific Study of Language", Cognition, 4, 321-398

15. Dresher, B. Elan and Norbert Hornstein (1977) "Reply to Schank and Wilensky", Cognition, 5, 147-149.

16. Dresher, B. Elan and Norbert Hornstein (1978) "Reply to Winograd", to appear in Cognition.

17. Fodor, Jerry (1975) The Language of Thought, Thomas Y. Crowell, New York.

18. Guttag, J. (1975) "The Specification and Application to Programming of Abstract Data Types", Technical Report CSRG-59, University of Toronto, Toronto, September 1975.

19. Hamilton, Margaret (1971) "Management of Apollo Programming and Its Application to the Shuttle", Software Shuttle Memo No. 29, The Charles Stark Draper Laboratory (hereafter cited as CSDL) Cambridge, Massachusetts.

20. Hamilton, Margaret (1972) "The AGC Executive and Its Influence on Software Management", Shuttle Management Note 2, CSDL, Cambridge, Massachusetts, February 1972.

21. Hamilton, Margaret and Saydean Zeldin (1973a) "Higher Order Software Requirements", Doc. E-2793, CSDL, Cambridge, Massachusetts, August 1973.

22. Hamilton, Margaret and Saydean Zeldin (1973b) "Higher Order Software Techniques Applied to a Space Shuttle Prototype Program", in Goos and Harmanis (eds.), Lecture Notes in Computer Science Vol. 19, Springer-Verlag, New York, pp. 17-31; presented at Program Symp. Proc. Colloque sur la Programmation, Paris, France August 1973.

23. Hamilton, Margaret and Saydean Zeldin (1976a) "Higher Order Software--A Methodology for Defining Software", IEEE Transactions on Software Engineering SE-2, 9-32.

24. Hamilton, Margaret and Saydean Zeldin (1976b) "The Foundations for AXES: A Specification Language Based on Completeness of Control", Doc. R-964, CSDL, Cambridge, Massachusetts, March 1976.

25. Hamilton, Margaret and Saydean Zeldin (1976c) "Integrated Software Development System/Higher Order Software Conceptual Description", TR-3, HOS, Inc., Cambridge, Massachusetts, November 1976.

26. Hamilton, Margaret and Saydean Zeldin (1976d) "AXES Syntax Specification", TR-4, HOS, Inc., Cambridge, Massachusetts, December 1976.

27. Hamilton, Margaret and Saydean Zeldin (1977) "The Manager as an Abstract Systems Engineer", Proceedings, COMPCON 77 Fall Conference, IEEE Computer Society, Washington, D.C., September 6-9, 1977.

28. Hamilton, Margaret and Saydean Zeldin (1978) "Discussion of an Appraisal of Program Specifications", to appear in Wegner, Dennis, Hammer and Teichrow (1978).

29. Hochstein, S. and R.M. Shapley (1976a) "Quantitative Analysis of Retinal Ganglion Cell Classifications", Journal of Physiology, 262, 237-264.

30. Hochstein, S. and R.M. Shapley (1976b) "Linear and Nonlinear Spatial Subunits in Y Cat Retinal Ganglion Cells", Journal of Physiology, 262, 237-284.

31. HOS (1977) "Techniques for Operating System Machines", HOS, Inc., Cambridge, Massachusetts, July 1977.

32. Levick, W.R. (1975) "Form and Function of Cat Retinal Ganglion Cells", Nature, 254, 659-662.

33. Miller, George (1976) "Semantic Relations Among Words: I", presented at MIT Convocation on Communications, Cambridge, Massachusetts, March, 1976.

34. Miller, George and P.N. Johnson-Laird (1976) Language and Perception, Harvard University Press, Cambridge, Massachusetts

35. Mills, H.D. and M.C. Wilson (1976) "An Introduction to the Information Automat", IBM, Gaithersburg, Maryland, May 7, 1976.

36. Parnas, D.L. (1972) "A Technique for Software Module Specification with Examples", Communications of the ACM, 15, 330-336.

37. Peters, Lawrence J. and Leonard L. Tripp (1977) "Comparing Software Design Methodologies", Datamation, November 1977, pp. 89-94.

38. Ramamorthy, C.V. and Siu-Vin F. Ho (1975) "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, SE-1, 1.

39. Richter, M.D. and J.D. Mason, (1976) "Software Requirements Engineering Methodology", TRW Defense and Space Systems Group, Huntsville, Alabama, 1 September 1976.

40. Robinson, L., K.N. Levitt, P.G. Neumann, A.R. Saxena (1975) "On Attaining Reliable Software for a Secure Operating System", Proceedings, International Conference on Reliable Software, Los Angeles, April 21-23, 1975.

41. Robinson, L., and K. N. Levitt (1977), "Proof Techniques for Hierarchically Structured Programs", Communications of the ACM, 20, 271-283.

42. Stevens, W.P., G.J. Meyers, and L.L. Constantine (1974) "Structured Design", IBM Systems Journal,

43. Wegner, Peter, Jack Dennis, Michael Hamner, and Daniel Teichroew (eds.), Proceedings of the Conference on Research Directions in Software Technology, October 10-12, 1977, to appear in Spring 1978, MIT Press, Cambridge, Massachusetts.

44. Werblin, Frank S. (1974) "Control of Retinal Sensitivity, II. Lateral Interactions at the Outer Plexiform Layer", The Journal of General Physiology 63, 62-87.

45. Wilson, M.L. (1976) "The Information Automat Approach to Design and Implementation of Computer-Based Systems", IBM, Gaithersburg, Maryland, April 1976.

46. Zilles, S.N. (1975) "Algebraic Specification of Data Types", Computation Structures Group Memo 119, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts.

PART 6

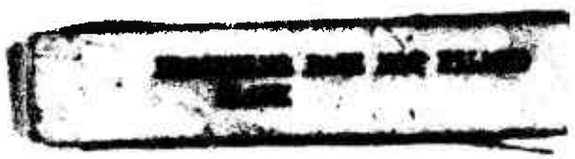
Lexical Functions and Lexical Decomposition: An Algebraic Approach to Lexical Meaning

by
S. Cushing

1. Functional Representation of Lexical Meaning

Research on lexical semantics in generative grammar, beginning with Katz and Fodor (1963), has traditionally incorporated two basic assumptions: first, that word meanings can be decomposed into primitive meanings and, second, that these primitive meanings are best represented in terms of semantic markers. The second of these assumptions is challenged by Miller (1976) and the first by Fodor, Fodor, and Garrett (1975). Miller suggests replacing semantic markers with a "functional representation" and Fodor, Fodor, and Garrett suggest replacing lexical decomposition with "meaning postulates". In this paper we examine Miller's proposal and derive some of its implications for the structure of the semantic lexicon. In particular, we derive a model for the lexicon in which word meanings are decomposed into functional primitives which are themselves characterized by something very much like meaning postulates, and we argue that such a model follows naturally from the use of "functional representation" for lexical meaning.

Miller questions "whether semantic markers, as they were defined by Katz and Fodor, are the best way to represent the concepts shared by different lexical entries" (p. 5) and proposes, instead, the use of "a functional representation of word meanings, especially when we go beyond nominal expressions to the more complex relations among verbs and prepositions." To illustrate his proposal, Miller gives the "semantic decomposition" of bring shown in Figure 1. He represents the "propositional information contained" (p. 7) in a sentence in which bring occurs by "BRING (x,y,), where x is a pointer assigned to" a bringer "and y is a pointer assigned to"



something brought. The five lines of Figure 1 then illustrate how "BRING (x,y)" might be broken down into specific combinations of other "functional representations," such as "COME (x)," "DO (x)", "CAUSE (x)", and "ACCOMPANY (x)".

> That is to say, bring would be decomposed into come, cause, and accompany. Come, in turn, would be decomposed into travel and speaker's location; cause would be decomposed into possible sequences of events, and accompany would be analyzeo into travel and with. Each of these, in turn, might be further decomposed. Since decomposition must stop somewhere, this program commits us to the assumption that there are certain semantic primitives into which all words can be analyzed.

"One assumes," Miller argues further, "that these primitives, whatever they are, are cognitive universals, independent of particular languages."

Since Miller suggests the use of a "functional representation", it seems reasonable to suppose that he indeed intends to be talking about functions. The use of "functional representation" for lexical meaning thus involves a qualitatively different kind of claim from that involved in the use of "semantic markers". Semantic markers are a kind of object that was invented specifically for the representation of lexical meaning; they have no significance outside of the theory in which they appear. Functions, in contrast, are an independently well-defined kind of mathematical object; to say that lexical meanings are, or can be represented as, functions automatically predicts a number of things about the behavior of lexical meanings. Equivalently, it forces us to ask a number of questions about lexical meaning that would not arise if semantic markers were to be used instead.

In this paper we examine some of the implications of the use of functional representation for lexical meaning, using Miller's decomposition of bring as an example. We will not be concerned here with the question of whether

or not the example provides the correct analysis of bring, and we will thus ignore many of the issues that would have to be dealt with in any attempt to answer that question; in particular, we will not ask whether "meaning postulates" might do a better job, by themselves, in characterizing the meaning of bring. The result of our investigation will be a proposed model for lexical meaning in which both "lexical decomposition" and "meaning postulates" play a role. We will argue that the semantic lexicon is a heterogeneous algebra, in the sense of Birkhoff and Lipson (1970), and that the study of lexical meaning properly involves determining the empirically correct characterization of such algebras for natural languages.

The logic here is a bit complex and a word of further clarification is perhaps in order. Katz and Fodor (1963), as well as Katz (1972, 1977), want semantic marker representation, semantic decomposition into primitives, and no meaning postulates. Miller (1976) concurs in wanting semantic decomposition into primitives and no meaning postulates, but wants to replace semantic marker representation with functional representation. Fodor, Fodor, and Garrett (1975) want meaning postulates instead of semantic decomposition into primitives, but take no position on representation, saying "all that we require is that formatives of the natural language should correspond to formatives in the representational system, whatever these latter may turn out to be" (p. 526). Since semantic markers were introduced specifically to represent semantic primitives, it seems reasonable to conclude that Fodor, Fodor, and Garrett reject that notation, but we cannot draw the further conclusion that they would necessarily accept functional representation, as Miller uses it, since other options, e.g., traditional logical predicate notation, are also available to them.

Hopefully, all of this will become somewhat clearer in what follows. We will be arguing, specifically, that the adoption of a functional representation for semantic decomposition leads one to an algebraic model for the lexicon which also involves "meaning postulates"; we will not examine the case in which semantic decomposition is represented in terms of semantic markers. It can be plausibly argued that semantic markers are themselves naturally interpretable as a form of functional representation, in which case they would also be covered by our argument, but this is not their intended interpretation (Katz, 1977, p. 582), and we will not deal with it here.[2]

SEMANTIC DECOMPOSITION OF BRING:

BRING(X,Y): COME(X) & DO(X,S) & CAUSE(S,ACCOMPANY(Y,X))

COME(X): TO(TRAVEL(X),SPEAKER)

DO(X,S): X DOES SOMETHING — PRIMITIVE CONCEPT

CAUSE(X,Y): HAPPEN(X) & HAPPEN(Y) & NOTPOSSIBLE(Y & NOT(X)) & BEFORE (X,Y)

ACCOMPANY (X,Y): WITH(TRAVEL(X),Y))

Figure 1

Miller's Decomposition of Bring

2. Miller's Functions: Domains and Ranges

A function is a many-one correspondence between two sets. The first set, called the domain, provides the function with its arguments or inputs; the second set, called the range, provides the function with possible values or outputs. Every member of the domain must be associated by the function with exactly one member of the range, but not every member of the range need be associated with a member of the domain; a range member may be associated with more than one domain member but need not be. If D and R are the domain and range, respectively, of a function f, then we write

(1) $f : D \rightarrow R$

to mean that f provides an association or mapping from D into R. For a function to be well-defined, its domain, its range, and its mapping must be clearly specified in some way.

The intended mappings of Miller's functions are presumably supposed to be given by the decomposition, and we will return to this question later. About their domains and ranges, however, Miller says nothing, except to remark, as we have noted, that by "SPEAKER" he means "speaker's location" and that cause is meant to "be decomposed into possible sequences of events". It follows presumably, that "SPEAKER" is a constant symbol denoting some member of the set of locations and that "x" and "y" in the fourth line of the decomposition are variables denoting members of the set of events. Beyond this we will have to try to figure out what the domains and ranges of Miller's functions must be, based on whatever information we can draw out of his decomposition. Some

of our specific judgements will admit of alternate interpretations, given the extremely limited character of the proposed lexicon fragment we are examining, hut we can still get a good idea, by proceeding in this way, of the kinds of questions that must be asked and answered, if a functional representation is to make sense.

Henceforth, we will consistently denote functions by symbols whose initial letter is capitalized and sets by symbols that are capitalized throughout. This will make it easier tn keep track of which things are sets and which things are functions. We have noted that "x" and "y" in line 4 are intended by Miller to denote events, so we can take the domain of the function Happen to be the set of events, denoted here by the symhol "EVENTS", and we can take the domain of Before to be EVENTS $\times$ EVENTS, that is, the set of ordered pairs of events. Since an event is intrinsically something that happens, it would seem to make little sense to distinguish between an event x and the event that consists of the happening of x. If this is correct, then we have to take the range of Happen to be some set other than EVENTS, and the set of truth values or "booleans" seems to be the most reasonable candidate. This gives us

$$\text{Happen: EVENTS} \rightarrow \text{BOOLEANS}$$

as a description of the function Happen in the standard mathematical form (1). Similarly, it seems unnatural to consider x happening before y to be an event distinct from x and y and, again, BOOLEANS seems to be the most reasonable alternative for the range of Before. This gives us

$$\text{Before: EVENTS} \times \text{EVENTS} \rightarrow \text{BOOLEANS}$$

as a standard representation of the function Before.

The situation with Notpossible is a little more complicated. On the one hand, the output of Notpossible is taken as an input to one of the "&" functions, along with the outputs of Happen and Before. Since these latter outputs are all booleans, it seems reasonable that the output of Notpossible must also be a boolean, because we would not expect a conjunction function to take inputs from more than one set. We can conclude that the range of Notpossible is BOOLEANS and that

$\&_1$: BOOLEANS x BOOLEANS x BOOLEANS x BOOLEANS → BOOLEANS

is the standard description of the "&" function whose symbol connects the four main conjuncts of line 4.

The input of Notpossible has to be an event, however; Miller explicitly tells us that he means to be talking about "possible sequences of events", as we have seen, and, in any case, events seem intuitively to be the most natural sort of thing to be predicating nonpossibility of. It follows that y & Not (x) must be an event and, since we expect "&" to denote a function that takes only inputs from the same set, it follows further that the value of Not (x) must be an event, in order for it to be conjoinable with y. This then gives us

Not: EVENTS → EVENTS

$\&_2$: EVENTS x EVENTS → EVENTS

Notpossible: EVENTS → BOOLEANS

as our standard description of the functions Not, Notpossible, and the & that appears in the scope of Notpossible, and

(2) Cause: EVENTS x EVENTS → BOOLEANS

as a description of Cause itself. We see that the meaning of Miller's line 4 requires us to distinguish two very distinct conjunction functions, a fact that is in no way discernable from its form alone.

Now that we have Cause figured out, we can attempt to perform a similar analysis of line 1, in which the symbol "Cause" appears. Since the range of Cause is BOOLEANS and since we want "&" functions to take inputs from only one set, we have to conclude that the ranges of Come and Do are also BOOLEANS. This immediately gives us

$$\&_3: \text{ BOOLEANS x BOOLEANS x BOOLEANS} \rightarrow \text{BOOLEANS}$$

as a description of the "&" function that appears in line 1. $\&_3$ and $\&_1$ can be reduced to a single binary "&" function, but only if we introduce extra parentheses in lines 1 and 4 or, equivalently, add the further stipulation that that & is associative. As Miller's decomposition stands, however, we must distinguish $\&_1$ and $\&_3$ as separate functions and, in any case, we must distinguish both of them from $\&_2$.

The symbols "x" and "y" in line 1 cannot denote events, as they do in line 4, because events are not the sort of thing that can either bring or be brought, at least in the most natural interpretation of these words. In line 1 these symbols presumably denote people or things, since people and things are what bring other people and things, in the usual sense of "bring". This gives us

$$\text{Bring: } (\text{PEOPLE} \cup \text{THINGS}) \times (\text{PEOPLE} \cup \text{THINGS}) \rightarrow \text{BOOLEANS}$$

$$\text{Come: } \text{PEOPLE} \cup \text{THINGS} \rightarrow \text{BOOLEANS}$$

as standard descriptions of Bring and Come or, taking "entity" to mean either a person or a thing,

(3) Bring: ENTITIES x ENTITIES → BOOLEANS

(4) Come: ENTITIES → BOOLEANS

as equivalent descriptions of these functions.

This leaves us with only Do and Accompany still to figure out in line 1. We have already seen that Cause takes two events as input, so the s that appears as an input to Cause in line 1 must be an event. This gives us

Do: ENTITIES X EVENTS → BOOLEANS

as a description of the function Do. Since the output of Accompany in line 1 is an input to Cause, it follows from (2) that it must be an event. The range of Accompany is thus EVENTS. The inputs of Accompany in line 1, however, are the same y and x that serve as the inputs of Bring, albeit in the opposite order, so they must both be entities, as indicated by (3). This gives us

(5) Accompany: ENTITIES X ENTITIES → EVENTS

as a description of the function Accompany, completing our analysis of line 1.

We now have descriptions in the standard form (1) of Bring, Come, Do, Cause, Accompany, Happen, Notpossible, Not, Before, and our three versions of &; only To, Travel, and With remain to be analyzed. Since the output of To in line 2 is the same as that of Come, we know that the range of To must be BOOLEANS because of (4). Since the output of With in line 5 is the same as that of Accompany, we know that the range of With must be Events, because of (5). We also know that "y" in line 5 denotes an entity, again because of (5), and that "SPEAKER" in line 2 denotes a location. This leaves us with only the domain and range of Travel still to be determined.

The domain of Travel has to be ENTITIES, because "x" in line 2 denotes both the input of Travel and the input of Come, which (4) tells us must be an entity. This conclusion is confirmed by line 5, because "x" in line 5 denotes both the input of Travel and the first input of Accompany, which (5) tells us must be an entity. There is little we can say about the range of Travel, however. Whatever the output of Travel is, it has to serve both as an input, along with a location, to To to produce a boolean and as an input, along with an entity, to Accompany to produce an event. Lots of possibilities suggest themselves and, in the absence of further evidence, no convincing conclusions can be drawn. Under the circumstances, we might just as well settle, with some intuitive plausibility, on activities as the sort of object that can serve as the output of Travel, though perhaps events might make just as much sense. This gives us

(6) To: ACTIVITIES x LOCATIONS → BOOLEANS

With: ACTIVITIES x ENTITIES → EVENTS

Travel: ENTITIES → ACTIVITIES

as the standard descriptions of To, With, and Travel, subject to the proviso that ACTIVITIES could just as well be replaced by EVENTS or some other set, as far as the evidence provided by this fragment is concerned.[3]

This gives us descriptions of the domains and ranges of all the functions that Miller includes in his proposed semantic decomposition of bring, derived from the detailed functional structure of the decomposition itself. Our results leave many questions unanswered, not the least of which is that of whether Miller's proposed decomposition actually provides the correct account of the meaning of bring. It seems strange intuitively, for example, to say that someone can do an event, but this is what Miller's decomposition

requires, as we have seen. Similarly, we have seen that, while both come and accompany are verbs, the range of Come must be BOOLEANS but that of Accompany must be EVENTS; otherwise the particular combination of functions in line 1 will make no sense. Similarly, both to and with are prepositions, but the range of To is BOOLEANS and that of With is EVENTS, for essentially the same reason. These apparent discrepancies may reflect the genuine meaning differences between the members of these pairs of words: Come is something one simply does, while accompany is something one does to someone else; to relates one to a location, while with relates one to another "entity". More likely, however, the discrepancies may be simply an artifact resulting from the extremely limited nature of the fragment we have been restricting ourselves to. A more extensive fragment might reveal that verbs like come and accompany each have both boolean and event-valued functions associated with them. Perhaps there is a universal functional operator provided by general semantic theory that automatically provides us, for each such event-valued function, with the corresponding boolean-valued function, so we do not have to state the duality of such verbs in the lexicons of particular languages. Lots of other possibilities suggest themselves and we need not enumerate them here.[4]

Clearly such questions can be resolved only by examining further data. Jackendoff (1978), for example, uses syntactic evidence from English to argue, in effect, that prepositions like on should be analyzed not as mapping pairs of entities to booleans as suggested by the analysis of Miller and Johnson-Laird (1976), but as mapping entities to locations. We would

thus get

On: ENTITIES → LOCATIONS

rather than

On: ENTITIES x ENTITIES → BOOLEANS

as the standard description of the function On. Similarly, in place of the description

To: ACTIVITIES x LOCATIONS → BOOLEANS,

which we gave in (6) for To, Jackendoff's arguments suggest

To: ENTITIES → PATHS

as the standard description.

The point is that there really is something to argue about here: these are significant questions, and empirical evidence can be brought to bear in trying to answer them. As we noted in Section 1, functions, unlike semantic markers on their usual interpretation, are an independently well-defined kind of mathematical object and thus provide us with a natural handle with which to grasp onto the meanings we are using them to model. The use of functional representation commits us to asking some very specific kinds of questions about lexical meanings, as we have tried to illustrate here. To the extent that the answers to these questions provide insight into the interesting facts of lexical meaning, they support the use of functional representation as the correct way to deal with it. If attempts to answer these questions yield no interesting insights or lead us astray and create confusion, then functional representation itself will have to be rejected. In either case, the questions we have begun asking here illustrate

the kinds of questions that have to be asked and answered in order for functional representation to make sense at all. Without domains and ranges there are no functions; with them there is at least something to discuss.

3. Sets, Functions, and Algebras

In the last section we derived a general picture of the semantic lexicon that looks something like the following: The semantic lexicon consists of a collection Σ of sets, including at least the sets listed in (7), and a collection ω of functions, including at least the functions listed in (8), where the ranges of the members of ω are members of Σ and the domains of the members of ω are Cartesian products of the members of Σ. The

(7) Σ: EVENTS

BOOLEANS

ENTITIES (= PEOPLE U THINGS)

ACTIVITIES

LOCATIONS

PATHS (if Jackendoff is right)

(8) ω: Happen: EVENTS → BOOLEANS

Before: EVENTS x EVENTS → BOOLEANS

$\&_1$: BOOLEANS x BOOLEANS x BOOLEANS x BOOLEANS → BOOLEANS

Not: EVENTS → EVENTS

$\&_2$: EVENTS x EVENTS → EVENTS

Notpossible: EVENTS → BOOLEANS

Cause: EVENTS x EVENTS → BOOLEANS

$\&_3$: BOOLEANS x BOOLEANS x BOOLEANS → BOOLEANS

Bring: ENTITIES x ENTITIES → BOOLEANS

Come: ENTITIES → BOOLEANS

Do: ENTITIES x EVENTS → BOOLEANS

Accompany: ENTITIES x ENTITIES → EVENTS

To: ACTIVITIES x LOCATIONS → BOOLEANS
(or To: ENTITIES → PATHS, if Jackendoff is right)

With: ACTIVITIES x ENTITIES → EVENTS

Travel: ENTITIES → ACTIVITIES

On: { either ENTITIES x ENTITIES → BOOLEANS (Miller and Johnson-Laird)
{ or ENTITIES → LOCATIONS (Jackendoff)

members of Σ are presumably the semantic types of the language, that is, (mental representations of) the sorts of objects the speaker/hearer takes to exist. The members of ω then constitute what the speaker/hearer says or understands about those objects, in some general sense of those notions.

Such a dual collection of sets and functions is exactly what is meant in mathematics by an algebra (Birkhoff and Lipson, 1970). An algebra is an ordered pair $[\Sigma, \omega]$, where Σ is a collection of sets and ω is a collection of functions whose ranges are members of Σ and whose domains are Cartesian products of the members of Σ. If Σ contains exactly one member, the algebra is said to be homogeneous, while if it contains two or more members, the algebra is said to be heterogeneous. To the extent that the full semantic lexicon can be obtained by simply adding more sets to the list in (7) and more functions to the list in (8), it follows that the semantic lexicon is itself an algebra and, in particular, a heterogeneous one, as we noted at the end of Section 1.

4. Axioms, Polynomials, and Meaning Postulates

Classes of algebras are typically characterized by providing constraints on the members of Σ and ω. Most commonly, constraints on the members of Σ take the form of requiring some of them to contain distinguished members, while constraints on the members of ω take the form of statements, called axioms, which require the equality of specified combinations of some of them. The term "variety" is also used in the sense of our "class" and "law" in the sense of our "axiom" (Barnes and Mack, 1975, p.7).

The most familiar class of algebras, for example, is probably that of the groups. In these algebras, Σ consists of a single set G that contains a distinguished neutral member, denoted here by "Neut", and ω consists of two functions, a group multiplication, denoted here by "Mult", and a group inverse, denoted here by "Inv", where Mult and Inv satisfy

(9) Mult: $G \times G \to G$

Inv: $G \to G$

An algebra [{G}, {Mult, Inv}] that satisfies (9) qualifies as a group, if there is a member Neut of G such that the following four axioms are satisfied for every member g, g_1, g_2, g_3, of G:

(10) 1. Mult $(g_1, \text{Mult}(g_2, g_3)) = \text{Mult}(\text{Mult}(g_1, g_2), g_3)$

2. Mult (g, Neut) = g

3. Mult (Neut, g) = g

4. Mult (g, Inv(g)) = Neut.

Axiom 1 says that the function Mult is associative, that is, that the order in which Mult is performed is irrelevant to its results (though the order of its inputs might still be relevant). Axioms 2 and 3 say that Neut is indeed

the neutral member of G in the sense that applying Mult to Neut and any group member g in either possible order leaves g unchanged. Axiom 4 says that Inv is in fact an inverse operation, in the sense that applying Inv to any group member g and then applying Mult to g and that output always produces the neutral element. The class of groups is fairly large, in an intuitive sense; it includes the integers with Neut = 0, Mult = +, and Inv = 0-; the positive rational numbers with Neut = 1 Mult = x, and Inv = 1÷; and many other algebras.

Algebra descriptions like that expressed in (9) and (10) are generally viewed in mathematics as characterizing abstract structures, in this case the group structure; we can view them just as well, however, as characterizing kinds of objects, in this case group members, and sets of kinds of functions, in this case group multiplication and inverse (Hamilton and Zeldin, 1976b; Cushing, 1978). An object is a group member if it is a member of a set on which there are functions defined that satisfy the conditions we discussed in connection with (9) and (10); a pair of functions constitute a group multiplication and inverse if their domains and ranges are as specified in (9) and they satisfy the conditions involved in (10). The kinds of objects and functions are defined, in other words, in terms of their mutual interaction, as specified in axioms and in statements of the form (1).

It seems, then, that we can take Miller's semantic decomposition of bring as providing us with a proposed algebraic characterization of the kinds of objects listed in (7) and the class of kinds of functions listed in (8). What Miller is really giving us, in other words, is a description of a class of algebras $[\Sigma, \omega]$, where Σ is given by (7) and ω is given by (8). The axioms of these algebras are presumably the individual lines of his decomposition, as listed in Figure 1, with the colons interpreted as equality signs.

There is actually more going on here, however. One of the most basic notions defined in connection with the notion of algebras is the class of polynomial functions on an algebra. The polynomial functions on an algebra $[\Sigma, \omega]$ are, essentially,[5] those functions whose outputs can always be obtained by repeated application of the members of ω. If we take the group that consists of the integers with Mult = + and Inv = 0-, for example, as we discussed in connection with (9) and (10), then the class of multiples of group members provides examples of polynomial functions on that algebra; if g is a group member, then the functions defined by

$$2g = g + g$$

$$3g = (g + g) + g$$

$$4g = ((g + g) + g) + g$$

$$2g + 5(-g) = (g + g) + (((((-g) + (-g)) + (-g)) + (-g)) + (-g))$$

are all examples of polynomial functions on this group. Similarly, if we take the group that consists of the positive rational numbers with Mult = x and Inv = 1:, as we also discussed in connection with (9) and (10), then the powers of group members provide examples of polynomial functions on that algebra; if g is a group member, then the functions defined by

$$g^2 = g \times g$$

$$g^3 = (g \times g) \times g$$

$$g^4 = ((g \times g) \times g) \times g$$

$$g^2 \times (1 : g)^5 = (g \times g) \times (((((1 : g) \times (1 : g)) \times (1 : g)) \times (1 : g)) \times (1 : g)$$

are all examples of polynomial functions on that group.

The point about polynomial functions is that their existence as functions that can be legitimately applied to relevant members of Σ members is entailed by the existence of the functions they are iterations of. The existence of the function we have denoted by "3g" on our first group, for example, is assured by the existence of the function + on that algebra; the existence of the function g^3 on our second group is assured, similarly, by the existence of the function x on that algebra. Polynomial functions are essentially abbreviations for repeated applications of more basic functions; they can thus be eliminated altogether by replacing them explicitly with the repetitions they abbreviate. It follows that if, for some algebra $[\Sigma, \omega]$, some members of ω can be seen to be expressable as polynomial combinations of other members of ω, then they can be removed from ω without essentially altering the algebra. Only those functions that are independent of each other, in the sense that none can be expressed as polynomial combinations of the others, need be included in the ω of a particular algebra. The existence of the polynomial functions, as functions legitimately defined on Σ members, is then automatically assured.

We can see now that Miller's semantic decomposition of bring provides us not with the axioms of a class of algebras, but with definitions of a class of polynomial functions on those algebras, namely, the functions that give the meanings of bring, come, cause, and accompany. Accompany, for example, performs a genuine mapping from pairs of entities to events, as indicated in (8) by the formulation

Accompany: ENTITIES x ENTITIES → EVENTS,

but the particular mapping it performs happens to be the same, according to Miller, as that performed by the indicated combination of With and Travel. Come, similarly, performs a genuine mapping from entities to booleans, as indicated in (B) by the formulation

Come: ENTITIES → BOOLEANS,

but the particular mapping it performs could be accomplished just as well by applying To to the output of Travel and to the constant location Speaker. Bring could then be defined, itself as a polynomial function, entirely without the use of Accompany and Come, by the formula

(11) Bring (x, y) = To(Travel(x), Speaker) & Do(x, s) &
Cause (s, With(Travel (y), x)),

and this could be expanded further, using the polynomial expansion of Cause in Miller's line 4, to the formula

(12) Bring (x, y) = To(Travel(x), Speaker) & Do (x, s) &
(Happen(s) & Happen (With (Travel (y), x)) &
Notpossible (With (Travel(y),x) & Not(s)) &
Before(s, With(Travel(y),x)).

In terms of the mappings they say are performed by Bring from pairs of entities to booleans, as indicated in (B) by

Bring: ENTITIES x ENTITIES → BOOLEANS,

each of (11) and (12) is identical to Miller's line 1, even though neither (11) nor (12) mentions Come or Accompany and (12) does not mention Cause. We can thus get exactly the same function Bring, as a legitimate mapping from entity pairs to booleans, whether or not we include Come, Accompany, or Cause among the functions of our algebras.

It follows that we can get exactly the same set of functions by removing Bring, Come, Cause, and Accompany from (8) and treating these functions as polynomial functions defined by Miller's decomposition. This gives us a class of algebras $[\Sigma, \omega]$ in which (ignoring Jackendoff's arguments for the moment and sticking strictly with Miller's fragment)

(13) Σ = {EVENTS, BOOLEANS, ENTITIES, ACTIVITIES, LOCATIONS}

and

(14) ω = {Happen, Before, $\&_1$, Not, $\&_2$, Notpossible, $\&_3$, Do, To, With, Travel},

where the domains and ranges of the members of ω in (14) are given in (8) and where Bring, Come, Cause, and Accompany are defined as polynomial functions on the algebra by Figure 1. Since no member of ω in (14) can be expressed in terms of any others as a polynomial function (according to the information Miller provides), it makes sense to call the members of ω the primitive functions of the algebra.[6] Note that Miller himself characterizes Do as "primitive" as stated in Figure 1, but says nothing about the other members of ω in (14). Some of these other functions might be further decomposable as polynomial functions in a full specification of the semantic lexicon, but it is clear, as we have seen, that they are primitive with respect to this fragment.

Now that we have sorted out the primitive functions of Miller's class of algebras, which must be explicitly specified as members of ω, from the polynomial functions, whose existence as mathematical mappings from domains to ranges in the algebras is automatically guaranteed once the primitive functions are given, a serious deficiency immediately becomes apparent. The formulation in (13) gives us the kinds of objects our algebras are characterizing, (14) gives us the primitive functions we are taking as characterizing them, and Figure 1 gives us a set of polynomial functions that add nothing to that

characterization but can still be treated as if they were functions of the class of algebras. What is totally missing now, however, is a set of axioms for our class of algebras! Without axioms the members of ω are entirely arbitrary, except for the domain and range descriptions given in (8). We know, for example, that Do maps an entity and an event, according to (8), to a boolean and that Happen maps an event to a boolean, but we know nothing at all about which entities and events are mapped to which booleans, that is, about <u>which</u> mappings from entity-event pairs to booleans and from events to booleans "Do" and "Happen", respectively, are supposed to denote. Without a set of axioms to characterize our primitive functions, nothing about our purported class of algebras is well-defined. In order, in other words, for our proposed account of this fragment of the semantic lexicon as a class of algebras even to <u>be</u> such an account, we must supplement (13), (14) and Figure 1 with an appropriate set of axioms.

A natural candidate for one such axiom, for example, might be something like

(15) (Do (x,s) $\supset$ Happen (s)) = True

where "$\supset$" denotes entailment, which would itself have to be characterized as either a primitive or polynomial function from pairs of booleans to booleans, that is,

$\supset$: BOOLEANS x BOOLEANS $\rightarrow$ BOOLEANS.

The formulation in (15) says that if some entity does some event,[7] then that event happens, a reasonable enough assertion to the limited degree of empirical adequacy that our tiny fragment allows. Similarly, we might include in our list

an axiom like

(16) (Notpossible (x) ⊃ Not_1 (Happen (x))) = True,

where Not_1 differs from Not in (8) in that it maps booleans to booleans, rather than events to events:

Not_1: BOOLEANS → BOOLEANS.

The assertion in (16) says that if an event is not possible, then it does not happen, again a reasonable candidate for an axiom of our fragment.

Formulations like those in (15) and (16), we now observe, are exactly what have been called "meaning postulates" in the literature.[8] Logicians have traditionally distinguished between functions, which map inputs to outputs, and predicates, which simply "hold" of arguments,[9] so (15), (16) would normally be written as

Do (x, s) ⊃ Happen (s)

and

Notpossible (x) ⊃ Not_1 (Happen (x)),

perhaps preceded by universal quantifiers and/or an assertion marker. We can always treat predicates as boolean-valued functions, however, just as we can replace sets with their characteristic functions, and to do so is useful, as we have seen, because it makes possible a uniform algebraic treatment of word meanings. It follows that, even if Miller is correct in his view that word meanings can be decomposed into more primitive ones, he still needs something like meaning postulates to characterize the primitive meanings themselves.

This conclusion cannot be avoided, it should be stressed, by claiming that the primitives are to be characterized by explicit algorithms or that they are innate. To give an explicit algorithm for a function is simply to decompose it further, so the functions it is decomposed into become new primitives, themselves in need of axiomatic characterization (Hamilton and Zeldin, 1976b; Cushing, 1978).[10] Claiming that the primitives are innate (or "cognitive universals, independant of particular languages", as, we have seen, Miller suggests) also gets us nowhere, because our theory still owes us, in that case, an explicit account of what it is that is innate. What is it that is innate when Do is innate that is different from what is innate when Happen is innate? Again, axioms of some sort would seem to be in order.

We have concluded that Miller "still needs something like meaning postulates", but, in fact, our result is much deeper than that. The "meaning postulates" (15), (16) are really axioms of a very highly restricted form, and there is no a priori reason to assume that axioms must necessarily be so limited. Kinship terms, for example, seem particularly in need of a more complex treatment: we might argue, quite plausibly, that words like husband, wife, etc. express not relations, i.e., boolean-valued functions in our terms, as is generally assumed, but people-valued functions and, in particular,

Husband: WOMEN → MEN

Wife: MEN → WOMEN.

If this is the case, then axioms like[11]

(17) Husband (Wife(m)) = m

would seem to provide a more appropriate analysis than meaning postulates like (15), (16), which typically express true entailments between boolean-valued functions. Our real conclusion, then, is not that Miller needs meaning postulates, but that he need axioms, some of which may or may not be meaning postulates. Note that this need for axioms falls right out of the mathematics of functional representation, as Miller uses it: without axioms there is no class of algebras and thus no polynomial functions, so the decompositions are meaningless (Hamilton and Zeldin, 1976b; Cushing 1973). The specific forms of the axioms and, in particular, whether any or all of them are meaning postulates in the traditional sense is an empirical question, however, and the plausibility of (17) suggests strongly that meaning postulates alone will not suffice.

These results give the study of lexical semantics a scientific status similar to that of sentential syntax. A lexicon, like a grammar, is not simply a list, in this case of words and meanings, but a highly structured mathematical object, in this case an algebra. The really interesting problem of lexical semantics thus becomes that of determining the universal and language-specific constraints on lexical algebras, just as the basic problem of sentential syntax is generally taken to be that of determining the empirically motivated constraints on generative grammars. What sets do we have to recognize in a lexical algebra for a human language? What combinations of sets can exist as domains and ranges of lexical functions? What forms can axioms take and what forms are ruled out? These and similar questions enable us to raise lexical semantics from the level of mere taxonomy to that of a search for scientific explanation.[12]

In view of these results, it is not surprising that some of the most recent work on lexical semantics can be interpreted as implicitly attacking precisely this problem. Fodor, Fodor, and Garrett (1975) argue, for example, that "meaning postulates mediate whatever entailment relations between sentences turn upon their lexical content. That is, meaning postulates do what definitions have been supposed to do . . ." (p. 526). In our terms, as we have seen, this amounts to the claim, first, that all word meanings represent boolean-valued functions and, second, that all axioms are of the general form of which (15), (16) are instances. Jackendoff (1978), as we have also seen, disputes the first of these claims, arguing that some words, including prepositions like on, represent location-valued functions, rather than boolean-valued ones. Cushing (1976) agrees, in principle, with Jackendoff's claim but argues further that the lexicon must recognize not only sets of people, things, and locations (which he calls "places"), but also sets of times and possible worlds, not as special components of model-theoretic interpretation, as in Kripke semantics or Montague grammar, but as sets of the algebra just like any other![13] This last result dovetails nicely with Chomsky's independent observation that "possible-worlds semantics" seems to amount to nothing more than the lexical entry for the word "logically possible" and words definable directly in terms of it (personal communication).

These results are intriguing and suggest that the algebraic model should be further investigated. Note that we have clearly not proven the correctness of the model; we have looked at very little data and are thus in no position to draw what is an essentially empirical conclusion. Our argument has been an

entirely theoretical one: if Miller uses functional representation for lexical decomposition, then he also needs "meaning postulates", and the algebraic model is a natural result. The model is attractive because it enables us to approach the lexicons of natural languages in the same way that linguists customarily approach their grammars, as we have pointed out, namely, by looking for universal constraints on the relevant formal systems.[14] It naturally subsumes, furthermore, some of the more interesting questions that have arisen about the semantic lexicon in the recent literature, as we have also pointed out. The real test, however, will, of course, be to examine lots of actual data in light of the model and see what interesting insights are revealed.

Footnotes

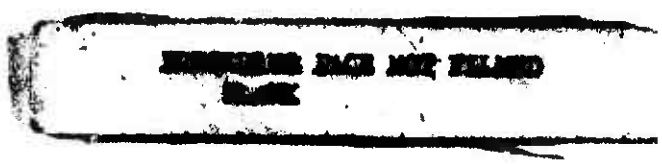
*Earlier formulations of the ideas in this paper were presented at various times and places under a number of different titles (Cushing 1977 a, b, c). A more general account, in which an attempt is made to incorporate the ideas discussed here into a proposed formal model for the study of cognition, is currently in preparation, based on the ideas in Cushing (1977b) and Cushing and Hornstein (1978). I would like to thank Mark Aronoff, Joan Bresnan, Dick Carter, Noam Chomsky, Morris Halle, Margaret Hamilton, George Miller, Michael Zeldin, Saydean Zeldin, and the members of the respective workshops for helpful discussion and encouragement.

1. Miller has confirmed this supposition (personal communication).

2. We will, however, interpret some of Jackendoff's (1973) results functionally, though he formulates them in terms of semantic markers. Whatever Jackendoff's own intent might be, the evidence he adduces and the conclusions he draws lend themselves naturally to the framework we will be developing here.

3. In a recent proposal for "a recursive scheme for describing the aspectual character of sentences" (p. 216), Steedman (1977) takes activities to be a kind of event, without using a functional representation for lexical meaning, however. Again, it is well beyond our present scope to inquire whether this assimilation of activities to events is correct.

4. A theoretically much more significant deficiency in Miller's analysis is created by the appearance of an "s" on the right-hand side of line 1 but not on the left. If bring itself is a function of two variables, it cannot be the same as a function of three variables. A set of general principles for avoiding errors like these is provided by Hamilton and Zeldin (1974, 1976a).



5. The actual definition is somewhat more complicated, of course. A concise formal definition of what he calls "polynomial operations" is given by Montague (1974, p. 224) for the case of homogeneous algebras, and extension to the heterogeneous case is straightforward. The difference between "function" and "operation" varies from author to author and, for our purposes, the two terms can be considered interchangeable. Polynomial functions constitute a special case of the "control-map" functions discussed by Hamilton and Zeldin (1976a, b). See also Cushing (1978).

   Note that heterogeneous algebras were not introduced until 1970 and thus were presumably not available to Montague. More significantly, Montague's use of algebras is very different from our own. As Thomason (1974, p. 48) points out, Montague was singularly uninterested in the lexicon, using algebras to give a formal explication of the notion "language" (1974, p. 225). Our concern here, however, is specifically with the lexicon and with its internal structure, in particular; it is the lexicon itself that we are suggesting is a heterogeneous algebra, whatever may be the case with language as a whole.

6. Hamilton and Zeldin (1976b) and Cushing (1977b, 1978) call these "primitive operations", but, as mentioned in note 5, the difference is entirely terminological, for our present purposes.

7. As we noted earlier, "does some event" is an unnatural usage, but it is motivated by Miller's fragment. Such discrepancies are precisely the sort of thing that we would expect further evidence to enable us to resolve.

8. See Fodor, Fodor, and Garrett (1975), Carnap (1956), Montague (1974), and Katz (1977), to mention just a few works in which meaning postulates are discussed.

9. Montague (1974, p. 305-306), for example, distinguishes between predicates and operations in precisely this way. See notes 5 and 6 on "function" vs. "operation".

10. One might argue that these algorithms decompose the semantic primitives into "perceptual" primitives of some kind, obviating the need for specifically semantic axioms. While it may be necessary, however, to relate the semantic primitives at some point to more fundamental brain structures (more on this in note 12), this need has nothing at all to do with the present problem. To require that all word meanings be decomposed directly into physiological structures, even assuming that this is possible (see Fodor (1975, Chapter 1) for a good refutation of simple reductionism of this kind), would be to give up the existence of a specifically semantic level altogether, including, in particular, the existence of semantic primitives, something Miller and other decompositionists certainly do not want to do. If something more abstract is meant by "perceptual", however, than the physiological mechanisms involved in perception, then it becomes very unclear what the difference is, in our present context, between "perceptual" and "semantic", other than terminology. Whether our abstract primitives are "semantic" or "perceptual", we still need axioms in order for decompositions like Miller's to make sense, and this is the point of our argument.

11. We assume, of course, for simplicity a strictly monogamous society in which everyone is married. Further axioms would naturally be needed to characterize these notions with full empirical adequacy. Note that the semantic types MEN and WOMEN are not motivated by the information in Miller's fragment, and we use them only for this example. The right to assume a meaning function for any word in the language seems to be implicit in Miller's analysis and follows explicitly from Fodor, Fodor, and Garrett's requirement "that formatives of the natural language should correspond to formatives in the representational system", once we adopt a functional representation; establishing the existence of a semantic type, however, presumably requires some argument, as Jackendoff's discussion suggests.

12. We have argued that the semantic lexicon is a heterogeneous algebra, but have phrased our discussion of axioms and polynomial functions in terms of classes of algebras. Any algebra description of the sort we have discussed will always have infinitely many possible implementations in terms of actual sets in Σ and functions in ω; in some cases these will

all be isomorphic with respect to the functions of the algebra but in some cases they will not. This may at first seem like a disturbing indeterminacy, but, in fact, it is exactly what we should expect of formal systems that are taken to model biological phenomena. If we assume that a model for the semantic lexicon is a model for a part of the mind and thus of the brain, then specifically semantic criteria can carry us only so far in characterizing it. (Chomsky (1975) develops a similar argument with respect to linguistic criteria and grammars.)

Suppose, for example, obviously contrary to fact, that semantic evidence leads us to conclude that the entire semantic lexicon can be described by the algebra description we gave for the groups. Suppose further, again contrary to fact, that the class of algebras characterized by that description contains exactly two members, which differ in how the set and functions are actually implemented in the brain. While semantic evidence would be unable to choose between these two algebras as the actual brain representation, we could still try to decide between them on non-semantic, e.g. physiological, grounds. It might be the case, for example, that only one of the two algebras is efficiently representable in the DNA code or in terms of neurocellular structures, in which case the other could be reasonably ruled out, despite the fact that it satisfies the relevant semantic criteria. (The DNA example was suggested by Michael Zeldin (personal communication).) The point here is that semantics, like the rest of linguistics, presumably deals with people and that all sorts of evidence must be brought to bear in developing a complete model of the human system of which the semantic lexicon is only a part. (This is another difference between us and Montague, by the way. See note 5 and Thomason (1974).)

13. An account of how quantifiers can be dealt with in this framework is also suggested by the discussion in Cushing (1976).

14. The notion that different formal systems underlie sentential syntax and lexical semantics, i.e., grammars vs. algebras, is plausible on psychological grounds "in that there is evidence . . . that the process of lexicalization is distinct from grammar acquisition" (Anderson, 1977, p. 133). Naturally, we would expect the two systems to interact, however, as Jackendoff's arguments suggest.

References

Anderson, John R. (1977) "Induction of Augmented Transition Networks", Cognitive Science, 1, 125-157.

Barnes, Donald W. and John M. Mack (1975) An Algebraic Introduction to Mathematical Logic, Springer-Verlag, New York.

Birkhoff, Garrett and John D. Lipson (1970) "Heterogeneous Algebras," Journal of Combinatorial Theory, 8, 115-133.

Carnap, Rudolf (1956) "Meaning Postulates", in Meaning and Necessity, University of Chicago Press, Chicago Illinois.

Chomsky, Noam (1975) Reflections on Language, Pantheon Books, Random House, New York.

Cushing, Steven (1976) The Formal Semantics of Quantification, UCLA Doctoral dissertation, available from the Indiana University Linguistics Club, Bloomington, Indiana.

Cushing, Steven (1977a) "Lexical Decomposition and Lexical Algebra", presented to the MIT Workshop on Language and Cognition, February 1977, Cambridge, Massachusetts.

Cushing, Steven (1977b) "Lexical Functions and Lexical Algebra: A Software Model for the Semantic Lexicon", presented to the MIT Workshop on Lexical Representation, December 1977, Cambridge, Massachusetts.

Cushing, Steven (1977c) "An Algebraic Approach to Lexical Meaning", presented at the Annual Meeting, Linguistic Society of America, December 1977, Chicago, Illinois.

Cushing, Steven (1978) "Algebraic Specification of Data Types in Higher Order Software (HOS)", to appear in the Proceedings, Eleventh Annual Hawaii International Conference on System Sciences, January 1978, Honolulu, Hawaii.

Cushing, Steven and Norbert Hornstein (1978) "Software Systems, Language, and Empirical Constraints", to appear in The Behavioral and Brain Sciences, 1, 1.

Fodor, Jerry A. (1975) The Language of Thought, Crowell, New York.

Fodor, J. D., J.A. Fodor, and M.F. Garrett (1975) "The Psychological Unreality of Semantic Representations," Linguistic Inquiry, 6 515-531.

Hamilton, Margaret and Saydean Zeldin (1974) "Higher Order Software Techniques Applied to A Space Shuttle Prototype Program", in B. Robinet (ed.), Programming Symposium, Proceedings, Colloque sur la Programmation, Paris, April 9-11, 1974, Vol. 19 of G. Goos and J. Hartmanis (eds.), Lecture Notes in Computer Science, Springer-Verlag, Berlin.

Hamilton, Margaret and Saydean Zeldin (1976a) "Higher Order Software--A Methodology for Defining Software", IEEE Transactions on Software Engineering, SE-2, 9-32.

Hamilton, Margaret and Saydean Zeldin (1976b) "AXES Syntax Description", TR-4, Higher Order Software, Inc., Cambridge, Massachusetts.

Jackendoff, Ray (1978) "Grammar As Evidence for Conceptual Structure," to appear in M. Halle, J. Bresnan, and G. A. Miller (eds.), Linguistic Theory and Psychological Reality, MIT Press, Cambridge, Massachusetts.

Katz, Jerrold J. (1972) Semantic Theory, Harper and Row, New York.

Katz, Jerrold J. (1977) "The Real Status of Semantic Representations," Linguistic Inquiry, 8, 559-584.

Katz, Jerrold, J. and Jerry A. Fodor (1963) "The Structure of a Semantic Theory," Language, 39, 170-210.

Miller, George A. (1976) "Semantic Relations Among Words: I," presented at the MIT Convocation on Communications, March 1976, Cambridge, Massachusetts.

Miller, George A. and Philip N. Johnson-Laird, Language and Perception, Harvard University Press, Cambridge, Massachusetts.

Montague, Richard (1974) Formal Philosophy, Yale University Press, New Haven.

Steedman, M.J. (1977) "Verbs, Time, and Modality," Cognitive Science, 1, 216-234.

Thomason, Richmond (1974) "Introduction" to Montague (1974).

Part 7

A NOTE ON ARROWS AND CONTROL STRUCTURES: CATEGORY THEORY AND HOS

S. Cushing

A NOTE ON ARROWS AND CONTROL STRUCTURES: CATEGORY THEORY AND HOS

"Nature is a structure of evolving processes. The reality is the process."

- A. N. Whitehead

In this note we briefly review the "arrow-language" (Arbib and Manes, 1975) of category theory and its use in defining some basic notions of set theory. We then use these results to shed new light on some basic notions of Higher Order Software (HOS) (Hamilton and Zeldin, 1976b).

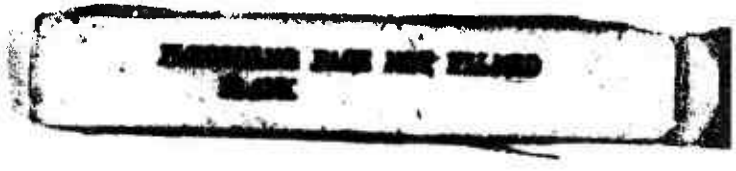
1. Arrows and Commutative Diagrams

The mathematical theory of categories is an attempt to develop a universal framework through which to unite the many seemingly different branches of mathematics. In the words of Arbib and Manes (1975),

> Category theory is the mathematician's attempt to lay bare some of the underlying principles common to diverse fields in the mathematical sciences. It has become, as well, an area of pure mathematics in its own right. Briefly, a category is a domain of mathematical discourse characterized in a very general way, and category theory is thus an array of tools for stating results which can be used across a wide mathematical spectrum (p. xi).

Underlying category theory is the idea that many of the definitions of fundamental mathematical notions can be reformulated in terms of mappings from one set to another, rather than in terms of the elements that sets contain. As Arbib and Manes put it,

> the usual approach to set theory starts with elements and builds all its notions in terms of these...we introduce a different approach to set theory, which builds all its notions in terms of arrows, the symbols $f: A \to B$ which represent a function as a unitary whole rather than in element-by-element terms...this 'arrow-language' of category theory allows us to specify, once and for all, concepts which play an important role in many different areas of mathematics even though their element-by-element definitions are drastically different in different domains of discourse (p. 1).



In this section, we review this arrow language, basing our discussion fairly closely on that of Arbib and Manes, and in the next section we apply it to HOS.

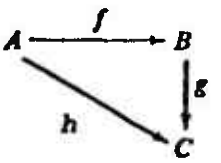
A mapping (function), in this arrow language, is represented by an arrow which points from (the symbol denoting) the domain of the mapping to its range (or co-domain). If a mapping f maps set A to set B, it is thus denoted by the symbol

$$A \xrightarrow{f} B.$$

Composition of mappings, i.e., application of one mapping after another has been applied, is denoted by a succession of such arrows. If f maps A to B and g maps B to C, we denote this fact by the symbol

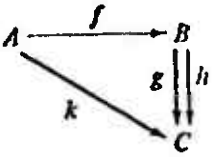
$$A \xrightarrow{f} B \xrightarrow{g} C.$$

If the composition of two mappings has the same effect as a single third mapping, we can denote this fact by a single diagram in which the arrows for all three mappings appear, as follows:



(1)

Such a diagram is called a commutative diagram; it is said to commute, because both paths from A to C have the same effect, i.e., any element of A maps to the same element of C whether we apply h alone or first f and then g. In general, commutativity is taken to hold over two or more paths only if at least one contains at least two arrows, such as the f, g path in (1). The diagram



(2)

for example, tells us that the f, g path; the f, h path; and the k path all produce the same result, since the first two contain two arrows, but it tells us

nothing about the relation between the g path and the h path, since each of these contains only one arrow.

This arrow language can be used to provide revealing reformulations of the definitions of many basic mathematical notions, as Arbib and Manes show. Of particular interest to us here is the definition they develop for the Cartesian product of two sets, which they call the product. Traditionally, the (Cartesian) product of two sets is defined as the set of all ordered pairs whose first element is in the first set and whose second is in the second. If A and B are two sets, their product A x B is thus given by

$$A \times B = \{(a,b) \mid a \in A \text{ and } b \in B\}.$$

Since this definition is formulated in terms of the elements of A and B, the question immediately arises, in our present context, whether it can be reformulated entirely in terms of mappings instead.

The first thing we notice in this connection is that there are two very special mappings associated with A x B, as follows:

$$\pi_1: \quad A \times B \to A, \quad (a,b) \mapsto a$$

$$\pi_2: \quad A \times B \to B, \quad (a,b) \mapsto b$$

The straight arrows indicate the domain-range relationships of the two mappings, while the barred arrows indicate the effect of the mappings on individual domain elements. The effect of π_1, in other words, is to map each ordered pair to its first component, while the effect of π_2 is to map each ordered pair to its second component. Arbib and Manes call these mappings projections, by analogy with analytic geometry, in which π_1 would amount to projection onto the abscissa and π_2 would amount to projection onto the ordinate.

Given any set C and any two mappings

$$p_1: \quad C \to A$$

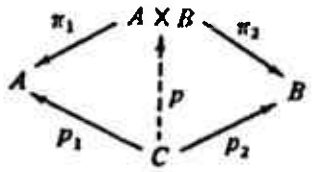
$$p_2: \quad C \to B,$$

we can clearly reverse, so to speak, the effects of π_1 and π_2 by defining a new mapping p as follows:

$$p: \; C \to A \times B$$

$$c \mapsto (p_1(c), \, p_2(c)).$$

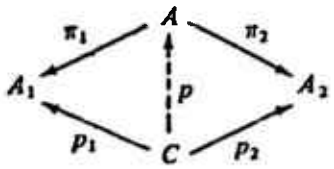
In other words, just as π_1 and π_2 take (a,b) to a and b, respectively, so p takes a and b to (a,b), where $a = p_1(c)$ and $b = p_2(c)$ for some c. What this means, however, is that the diagram



(3)

commutes, i.e., that any indicated path from one particular set to another has the same mapping effect as any other. As Arbib and Manes point out, there is, for any given A, B, C, π_1, π_2, p_1, p_2, only one p that will work in (3). Given this fact we can turn (3) into a definition of product, as follows:

A product of two sets A_1 and A_2 is a set A equipped with two maps $\pi_1 : A \to A_1$ and $\pi_2 : A \to A_2$ (called projections) with the property that, given any other set C with pair of maps $p_1 : C \to A_1$ and $p_2 : C \to A_2$, there exists a unique map p such that



(4)

Arbib and Manes show that, if there are two distinct products for two sets, then they must be isomorphic in the sense that one is just a relabelling of the elements of the other. For example,

$$\{(b,a) \mid b \in B \text{ and } a \in A\}$$

satisfies the definition of product just as well as

$$\{(a,b) \mid a \in A \text{ and } b \in B\}$$

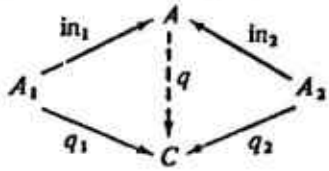
does. These two sets differ only in the way we are writing their elements, however,

in that each element of one can be obtained by reversing, i.e., relabelling, the components of the corresponding member of the other. It follows that we can talk about the product of two sets, ignoring differences between sets that are isomorphic in this sense.

This brings us to the notion of duality. Given a notion that has been defined in terms of commutative diagrams, we get the dual of that notion by reversing the directions of all the arrows in the diagram. It turns out, as Arbib and Manes discuss, that all theorems remain true in category theory when we replace every notion in the theorem with its dual. If P is a property, such as being the product of two sets, that has been defined in terms of commutative diagrams, then the property dual to P is called co-P. Proving a theorem of the form "All W's have property P," thus automatically provides a proof of a dual theorem of the form "All co-W's have property co-P." This leads us to ask the obvious question: Given our definition of the product of two sets, what is the co-product of two sets? In other words, what is the dual of the notion product?

Given the definition of "dual," we know that the co-product of two sets, whatever it is, is defined in exactly the same way as the product, except that the arrows in the relevant commutative diagram, i.e., (4), are reversed. This gives us the following definition:

A coproduct of two sets A_1 and A_2 is a set A equipped with two maps $in_1 : A_1 \longrightarrow A$ and $in_2 : A_2 \longrightarrow A$ (called injections) with the property that, given any other set C with a pair of maps $q_1 : A_1 \longrightarrow C$ and $q_2 : A_2 \longrightarrow C$, there exists a unique map q such that (5)



The definition tells us what a co-product is, but it doesn't tell us whether, for any two sets at all, any such thing exists. Whereas, for the product, we knew from the traditional definition what we were talking about and that it exists, and we derived the arrow definition from that, all we know from the arrow definition of co-product is that it satisfies (5), if it exists at all. By duality we know that all co-products of two sets are isomorphic, so we can talk

about the co-product of two sets, but we still do not know what the co-product of two sets looks like, aside from the role that it plays in (5).

As Arbib and Manes show, the co-product of two sets amounts to none other than their disjoint union. If two sets are disjoint, then their disjoint union is simply (isomorphic to) their union, but if they are not disjoint, then we first have to label their elements and then form the union in order to get their disjoint union. One way of labelling elements is to associate a different integer with each set, so, given two sets A_1 and A_2, we form the labelled sets

$$A_1 \times \{1\} = \{(a_1,1) | a_1 \in A_1\} \tag{6}$$

$$A_2 \times \{2\} = |(a_2,2)| a_2 \in A_2\}. \tag{7}$$

If an element a of A_1 is also an element of A_2, i.e., if A_1 and A_2 are not disjoint in the first place, then a appears as (a,1) in (6) and as (a,2) in (7), so (6) and (7) clearly are disjoint. If we then let $A_1 + A_2$ be the union of (6) and (7), i.e.,

$$A_1 + A_2 = (A_1 \times \{1\}) \cup (A_2 \times \{2\}),$$

then the resulting disjoint union of A_1 and A_2 clearly satisfies (5), with $A = A_1 + A_2$ and

$$in_k: \ A_k \to A_1 + A_2, \ \ a \mapsto (a,k), \ \ k = 1,2,$$

because then there is a unique q, namely,

$$q: \ A_1 + A_2 \to C, \ \ (a_k,k) \mapsto q_k(a_k), \ \ k = 1,2,$$

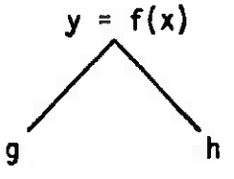
that works in (5). We see that the Cartesian product and the disjoint union of two sets are category-theoretic duals, in that their arrow-language definitions differ only in the directions of corresponding arrows. Clearly this is an intimate relationship that we would never have been able to guess at from the traditional definitions of these notions in terms of the elements of sets.

2. Systems and Control Maps

Higher Order Software (HOS) is a formal methodology for specifying computer-based systems in terms that are entirely independent of their implementation in hardware or resident software (Hamilton and Zeldin, 1976b). A system is said to be reliable if it produces the correct outputs from relevant inputs in the way it is supposed to. Formally, HOS is based on six axioms which guarantee the absence of interface errors among the components, or modules, making up a system and so help to ensure reliability in this sense (Hamilton and Zeldin, 1974).

Any system can be specified in HOS in terms of data types, functions, and control structures (Hamilton and Zeldin, 1976a). Data types are the kinds of objects that play a role as inputs or outputs in a system and its component subsystems; functions are the mappings among data types that get performed on these objects; and control structures are the relations that determine how the various functions in a system interact and combine to achieve the system's overall effect. In this note we will focus primarily on control structures, with only passing attention, as needed, to functions and data types.

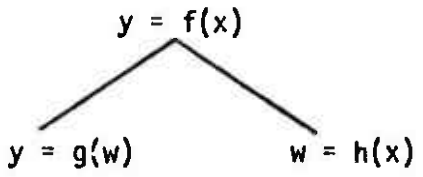
Given a system that performs a particular function f, it may be the case that the effect of f is really achieved by two (or more) other functions acting together. From the engineer's point of view, we might want to design our system to perform f by performing two other functions instead. If f is a function mapping input values x to output values y, and if g and h are functions which together produce exactly the same mapping from x to y as f, then we can express this relation in a tree diagram like the following:



(8)

Diagram (8) is an incomplete example of what in HOS is called a control map. Control maps specify the control relationships by means of which higher-level functions get performed through the action of functions at lower levels in the tree. To get a full control map, we would have to complete (8) by specifying in detail the relationships that hold among the inputs and outputs of f, g, and h, and, if we want, by further developing the tree by also expanding g or h into lower-level functions. Each such way of completing (8) is called a control structure and represents a unique relationship of data and control flow among the functions that make up the system. Note the intimate relationship involved here between data flow and control flow: the flow of control is specified in (8) by specifying the flow of data among various functions in the tree. Data flow and control flow can thus be viewed as "duals," since we cannot have one without the other and, in fact, we get one precisely by specifying the other. This is a very different sense of "dual" from the category-theoretic sense described in Section 1, however (see Cushing, 1977 for more discussion).

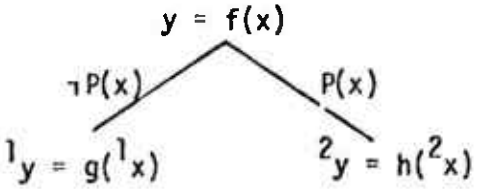
HOS recognizes three control structures as primitive, since any other control structure can be expressed in terms of those three. The composition primitive control structure is illustrated as follows:



(9)

In this control structure, one lower function takes the input of the higher function and produces an intermediate output which the other lower function takes as input to produce the output of the higher function. In (9), in other words, x gets mapped to y by f through first being mapped to w by h and then having w be mapped to y by g.

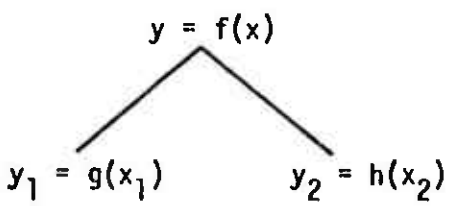
The set-partition primitive control structure is illustrated as follows:



(10)

In this control structure, the input values are divided into two distinct sets, according to whether or not they satisfy some predicate. Each lower function then maps one of these classes to output values independently of the other lower function. In (10), in other words, x is mapped to y by f through first determining whether x has the property P: if x has P, then it gets mapped to y by h, but if it does not have P, then it gets mapped to y by g. The left superscripts serve to underscore the fact that two distinct varieties of inputs and the outputs resulting from them are being distinguished in the tree.

The class-partition primitive control structure is illustrated as follows:



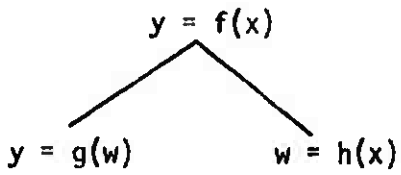
(11)

In this control structure, the input values themselves are seen as being made up of smaller units, e.g., vectors and their components, and the units are mapped one by one and independently by the lower functions to the units that make up the output values. In (11), for example, x values are seen as each consisting of both an x_1 value and an x_2 value, in that order, and these are mapped to y_1 and y_2, respectively, which make up y, by g and h, respectively. As in (10), but in contrast to (9), g and h work entirely independently of each other in (11); the difference between (10) and (11) is in the relationship borne by g and h to f in the two cases.

3. Control Maps and Commutative Diagrams

In the last section we outlined the basic HOS notions of control map and primitive control structure in more or less traditional mathematical terms, such as "inputs," "outputs," and the like. In this section, we ask: What happens if we try to reformulate these definitions in terms of the arrow language we discussed in Section 1? Answering this question amounts, first, to figuring out what basic mappings are involved in each primitive control structure and, then, expressing the relations among these mappings in terms of commutative diagrams.

The composition primitive control structure is easy, because it really amounts to the same thing as category-theoretic composition. This primitive control structure looks essentially like the following:



(12)

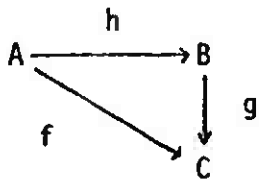
Each of the variables "x," "y," "w," takes on values belonging to some set (structured algebraically as a data type, c.f. Cushing, 1978). Let the "x" set be A, the "w" set be B, and the "y" set be C. What (12) tells us, first is that there are three distinct mappings involved in this primitive control structure, namely, f mapping A to C, g mapping B to C, and h mapping A to B. In arrow terms, we can write these mappings in the form

$$A \overset{f}{\to} C$$

$$B \overset{g}{\to} C$$

$$A \overset{h}{\to} B$$

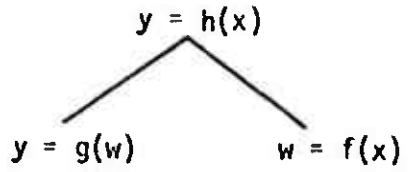
The control map also tells us, however, and this is key, that the joint effect of h followed by g has exactly the same effect as f alone; that is, if we want to perform f, we can do it just as well by forgetting about f and doing h and then g, instead. In arrow terms, this tells us that the diagram



(13)

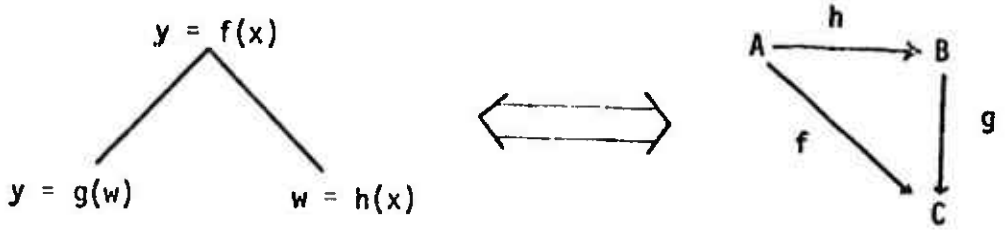
is commutative. Diagram (13), however, is exactly the same as (1), except for the labelling of the mappings. Diagram (13) describes the situation in which f can be eprformed by performing h and then g, instead, whereas (1) describes the situation in which h can be performed by performing f and then g, instead.

It follows that we can rewrite (1) as the HOS control map



(14)

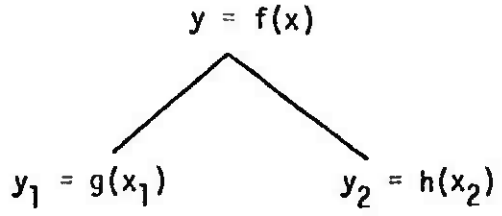
switching, in effect, the names of f and h. Clearly, for our present purposes, names are unimportant; what matters is the equivalence, in terms of the relations among the relevant mappings described in the respective diagrams, of (1) and (14). Diagram (12) is the same as (13), and (14) is the same as (1); in other words, the diagrams



(15)

are equivalent, as indicated, the first expressed in HOS terms, while the second is expressed in category-theoretic terms.

The class-partition primitive control structure is not so easy; its arrow-language interpretation is not obvious at all. Class-partition looks essentially like the following:



(16)

Now what does (16) tell us about the mappings f,g, and h? We know that f can be performed by performing g and h instead--this is what every control map tells us: higher-level functions can be performed by performing the immediately lower

ones instead--but this tells us nothing about how to write this fact in terms of arrows, since g and h are clearly not related by composition. Let us put this fact aside, temporarily, and see what else we can learn from (16).

We know from our discussion in section 2 that "x" in (16) takes on values that are complex relative to the values of "x_1" and "x_2": x_1 and x_2 are the units, in order, that make up x. The diagram thus presupposes two mappings-- call them d_1 and d_2, respectively--which map x values to their respective x_1 and x_2 component values. The values of "y," furthermore, are also complex, relative to the values of "y_1" and "y_2": y_1 and y_2 are the units that make up y. Two other unshown mappings are thus also presupposed by (16), a mapping--call it π_1--that maps y values to their y_1 values and a mapping--call it π_2--that maps y values to their y_2 values. If we again let A be the set of x values and C the set of y values--we do not need B, because there are no w values this time--and we now let A_1 and A_2 be the sets of x_1 and x_2 values, respectively, and C_1 and C_2 the sets of y_1 and y_2 values, respectively, then we see that (16) describes a situation involving the following seven mappings:

$A \xrightarrow{f} C$

$A_1 \xrightarrow{g} C_1$

$A_2 \xrightarrow{h} C_2$

} shown explicitly in the control map

$A \xrightarrow{d_1} A_1$

$A \xrightarrow{d_2} A_2$

$C \xrightarrow{\pi_1} C_1$

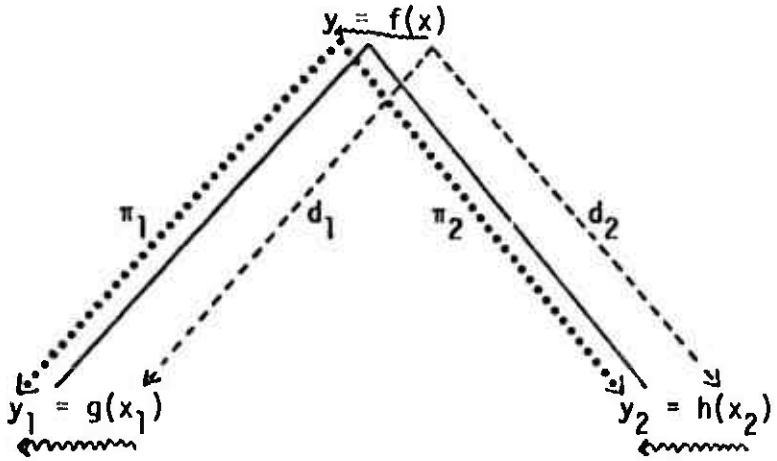
$C \xrightarrow{\pi_2} C_2$

} implicit in the notion "class-partition"

rather than just the three shown explicitly in the control map.

Now what does (16) tell us about the relationships among all these mappings? We have observed, and have temporarily put aside the fact, that f in (16) is performed by performing both g and h. This fact, however, is precisely what we need to answer our question about the seven mappings. If we put the four extra

mappings explicitly into the control map (16), we end up with a diagram like the following:



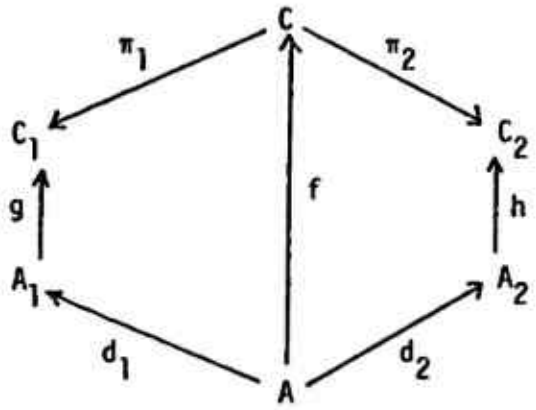
(17)

The two straight lines are from the original control map (16); the two dashed lines denote d_1 and d_2 from x to x_1 and x_2; the dotted lines denote π_1 and π_2 from y to y_1 and y_2; the three wavy lines denote the original mappings f, g, and h, from x to y, from x_1 to y_1, and from x_2 to y_2. These extra lines just make explicit what is implicit in the notion "class-partition" in (16). Note the directions of the arrows, however. There are only four ways, in (16), that we can complete full paths consisting of more than one arrow, the minimum number of arrows that we need to compare paths in a commutative diagram, as we saw in connection with (2). In (17), the four relevant paths are as follows:

$$
\begin{array}{ll}
\text{(a)} & x \overset{f}{\rightsquigarrow} y \overset{\pi_1}{\cdots\cdots\rightarrow} y_1 \\
\text{(b)} & x \overset{d_1}{\dashrightarrow} x_1 \overset{g}{\rightsquigarrow} y_1 \\
\text{(c)} & x \overset{f}{\rightsquigarrow} y \overset{\pi_2}{\cdots\cdots\rightarrow} y_2 \\
\text{(d)} & x \overset{d_2}{\dashrightarrow} x_2 \overset{h}{\rightsquigarrow} y_2
\end{array}
\qquad (18)
$$

What this tells us is that we can get from x to y_1, either by first performing f and then π_1, or by first performing d_1 and then g, and that we can get from x to y_2, either by first performing f and then π_2 or by first performing d_2 and then h. In other words, paths (a) and (b) in (18) perform exactly the same mappings from x values to y_1 values, and paths (c) and (d) perform exactly the same mappings from x values to y_2 values.

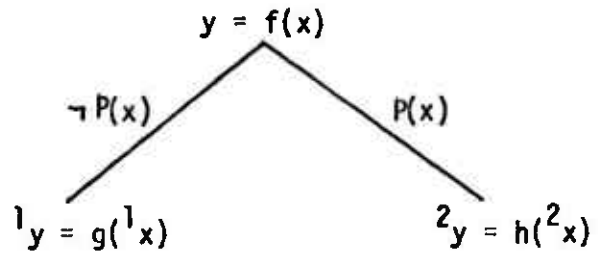
These equivalences give us the following commutative diagram, as an account of the relationships among the mappings that are explicitly and implicitly involved in (16):



(19)

To say that g and h are related to f by the class-partition primitive control structure is thus to say, at least in part, that there are mappings π_1, π_2, d_1, d_2 such that the diagram in (19) commutes.

Now what about the set-partition primitive control structure? Set-partition looks essentially like the following:



(20)

Again, we have to ask what (20) tells us about the mappings f, g, h, and, again, we have to observe that these are not the only mappings involved. The values of "^1x," "^2x," "^1y," "^2y" in (20) are not components of the values of "x" and "y", as are "x_1," "x_2," "y_1," "y_2" in (16). The values of "^1x," "^2x," "^1y," "^2y" in (20) are themselves whole values of x; the point about these values in set partition is that "^1x," "^2x," while themselves values of "x" in the domain, are also values of "x" in one of the sets that partitions the domain. In other words, if we let A_1 be the set of "x" values that satisfy "$\neg P(x)$," A_2 the set of "x" values that satisfy "P(x)," and A, again, the set of "x" values, altogether, then we have to distinguish between an "x" value a as a member of A and a as a member of A_1 or A_2, even though the same a is involved in both cases. If we let C, again, be the set of "y" values; C_1 the set of "y" values that come from "x" values in A_1; and C_2 the set of "y" values that come from "x" values in A_2, then, similarly, we have to distinguish between a "y" value c as a member of C and the same "y" value c as a member of C_1 or C_2. (Note that, while A_1 and A_2 must be mutually exclusive and exhaustive, C_1 and C_2 do not, since this will depend on the effects of f, g, and h.)

It follows that, as with class partition, there are more mappings involved implicitly in set partition than are shown explicitly in (20). Given the sets A, A_1, A_2, C, C_1, C_2, we have to distinguish, again, four mappings other than f, g, and h: a mapping i_1 from A_1 to A that takes each member of A_1 to itself as a member of A; a mapping i_2 from A_2 to A that takes each member of A_2 to itself as a member of A; a mapping j_1 that takes each member of C_1 to itself as a member of C; and a mapping j_2 that takes each member of C_2 to itself as a member of C. This gives us the following seven mappings:

$$A \xrightarrow{f} C$$

$$A_1 \xrightarrow{g} C_1$$

$$A_2 \xrightarrow{h} C_2$$

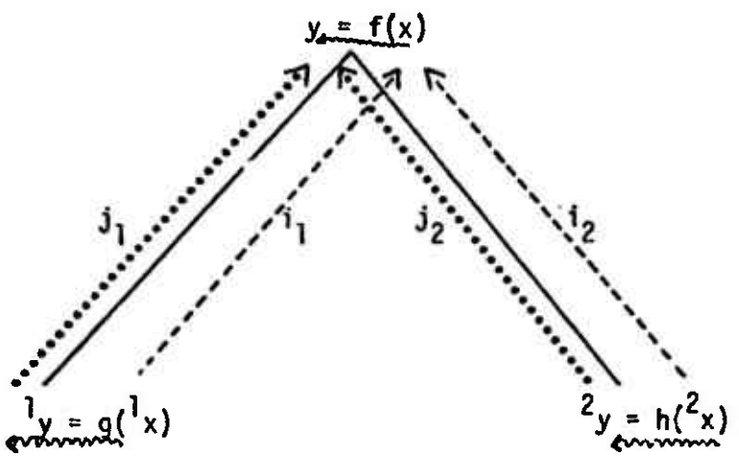
$$A_1 \xrightarrow{i_1} A$$

$$A_2 \xrightarrow{i_2} A$$

$$C_1 \xrightarrow{j_1} C \qquad C_2 \xrightarrow{j_2} C$$

as the mappings really involved in (20).

Again, we can determine the relationships among these mappings by putting them all explicitly into the control map, in this case (20). This gives a diagram that looks like the following:



(21)

Again, the straight lines are from the original control map, and the wavy lines denote the original mappings f, g, and h; the dotted lines, this time, denote the j mappings, and the dashed lines denote the i functions. If we note the directions of the arrows, we again see that there are only four ways of completing full paths of more than one arrow, as follows:

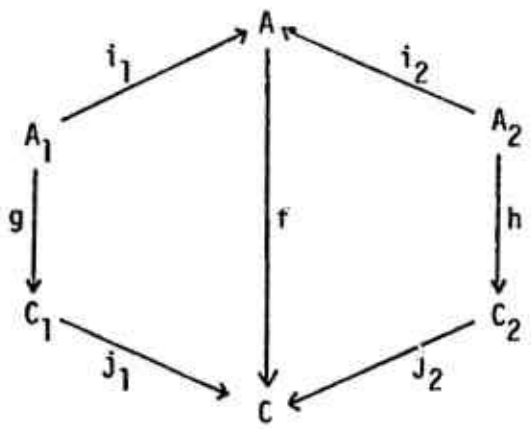
$$1x \xrightarrow{i_1} x \xrightarrow{f} y$$

$$1x \xrightarrow{g} 1y \xrightarrow{j_1} y$$

$$2x \xrightarrow{i_2} x \xrightarrow{f} y$$

$$2x \xrightarrow{h} 2y \xrightarrow{j_2} y$$

These relationships then give us the following commutative diagram, as an account of the relationships among the mappings involved in set partition:



(22)

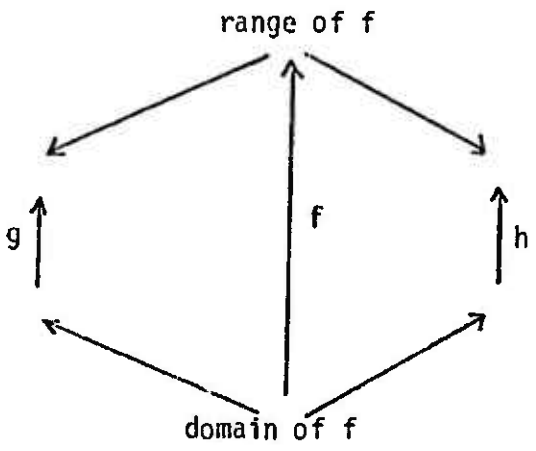
To say that g and h are related to f by the set-partition primitive control structure is to say, at least in part, that there are mappings i_1, i_2, j_1, j_2 such that the diagram in (19) commutes.

4. Primitive Control Structures and the Primacy of Process

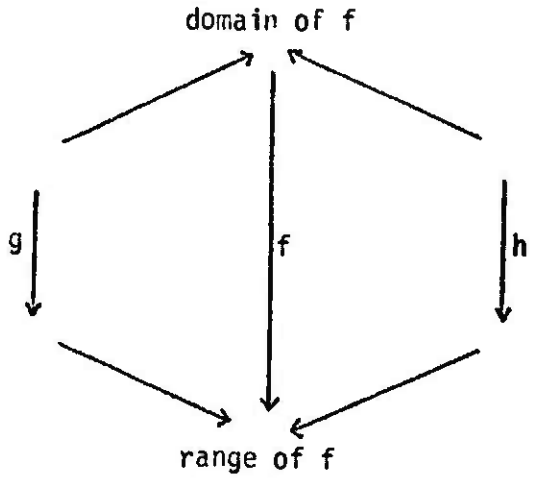
Diagrams (19) and (22) are interesting in their own right, because we said we wanted to determine how the definitions of the primitive control structures could be reformulated in arrow-language terms, and (19) and (22), along with the straightforward (15), provide the answer to this question. If we examine these two diagrams more closely, however, some further interesting facts emerge concerning the basic properties of and relations between the primitive control structures they represent.

If we ignore the labels for the various parts of (19) and (22), (clearly what we call things is not of fundamental importance), and compare the structures of the diagrams themselves (see (23)), a very surprising fact emerges.

Clearly, (19) and (22) are identical, except for the fact that the arrows are reversed! Since f points in opposite directions in the two cases, we might be led to suspect that either diagram could be transformed into the other by



[19]



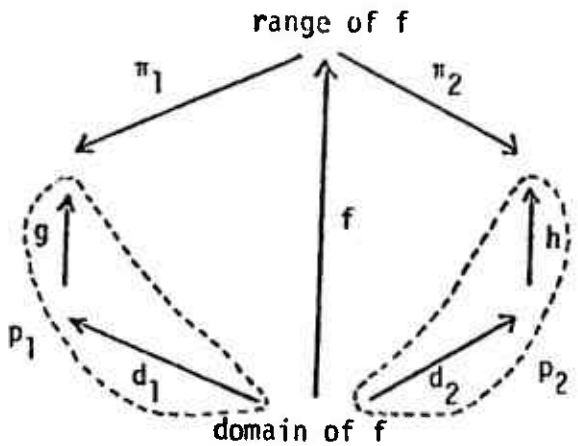
[22]

(23)

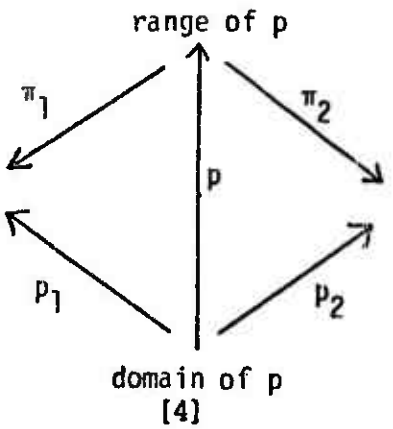
turning it upside down, but this is easily seen not to work. The difference between (19) and (22) is deeper than just the orientation of the diagrams on the page. In (19) all arrows other than f itself that touch either the domain or range of f point away from them, while all such arrows in (22) point toward them. The difference between the two diagrams is really solely a matter of the directions in which corresponding arrows point.

It follows that set-partition and class-partition are category-theoretic duals, as we defined this notion in Section 1, a fact that we would never have been able to guess at from (10) and (11) alone. Given any theorem about either set-partitions or class-partitions that is formulated entirely in terms of arrow-defined notions, we can automatically obtain a theorem about the other primitive control structure by simply replacing each arrow-defined notion by its category-theoretic dual. No further proof of a theorem obtained in this way would be necessary.

So much for comparing (19) and (22) themselves. What happens if we compare these two commutative diagrams to those we examined in Section 1. If we again ignore labels, which are entirely arbitrary, as long as our usage of them is self-consistent, then another interesting fact emerges:

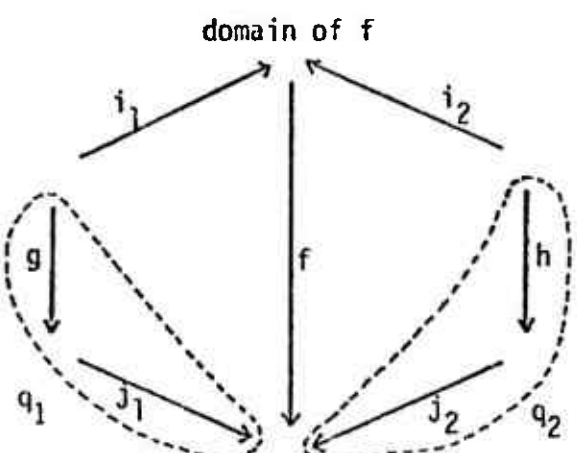


[19]

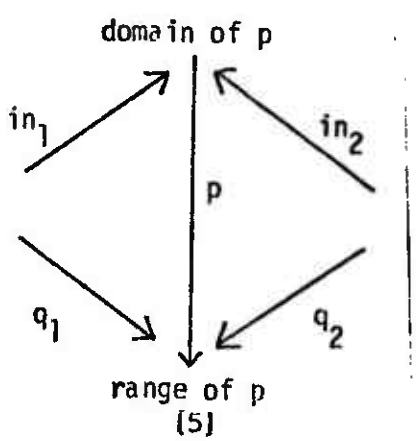


[4]

(24)



[22]



[5]

(25)

Diagram (19) is identical to (4) and diagram (22) is identical to (5), except that, in each case, each basic mapping in (4) or (5) is further broken up into two submappings related by composition. The only difference between (19) and (4), in other words, is that the arrow that represents p_1 in (4) is broken up into two composed arrows in (19), namely, g and d_1, and the arrow that represents

p_2 in (4) is broken up into the two composed arrows h and d_2 in (19); (22), similarly, differs from (5) only in that q_1 in (4) is written as g followed by j_1 in (22), and q_2 in (4) is written as h followed by j_2 in (22).

This difference is easy to explain in that Arbib and Manes are interested solely in defining the product and co-product A in (4) and (5) and are not interested in any properties C might have, other than its being a set, whereas, in HOS we want to partition both the domains and ranges of our functions, so that we can guarantee the traceability of data flow. The range in set-partition cannot always be partitioned exhaustively and in a mutually exclusive way, as can the domain and both the domain and range in class partition, but we still want to keep track of which outputs come from which partition member of the domain, as indicated in left superscripts on the variables in (10) and (20). Clearly, the effect of j_1 and j_2 in (22) (see (25)) is precisely to achieve this effect.

We see that the commutative diagrams that are implied by the class-partition and set-partition of functions in HOS are essentially the same as the diagrams that define the product and co-product of sets, respectively, in category theory. The single minor difference between the two sets of commutative diagrams reflects a slight difference in goals that is easy to understand and explain. There is more to this difference in goals than might be apparent from how it shows up in the diagrams, however. A deeper point emerges when we step back from the similarity of the commutative diagrams and examine the use to which the diagrams are being put in each case.

Arbib and Manes use (4) and (5) to define a set that is related to two given sets through the relation of being their product or co-product: a set A is the product or co-product of sets A_1 and A_2 if there is a unique mapping p or q, respectively, that makes (4) or (5), respectively, commute. The primitive control structures associated with (19) and (22), however, are used in HOS to define a function (mapping) that is related to two given functions through the relations of set-partition or class-partition: f is related to g and h by class-partition or set-partition if f is the unique mapping that makes (19) or (22), respectively, commute; clearly names do not matter, in this context, and we could just as well call f the product or co-product of g and h.

This difference between the category-theoretic use of (4) and (5) and the HOS use of (19) and (22) looks very much, it seems, like the difference between set-theory and category theory that Arbib and Manes outline in the passages cited in Section 1. Whereas "the usual approach to set theory" focuses on sets and the elements which make them up, category theory treats sets themselves as unitary wholes--elements, so to speak--and focuses on the mappings between them. Similarly, whereas Arbib and Manes use (4) and (5) to define sets, HOS uses the relations expressed in (19) and (22) to define mappings. In this sense, then, what HOS seems to do is to take the basic idea of category theory--i.e., shifting emphasis from sets of elements to mappings--and carry it one step further.

Its emphasis on the functions that systems perform seems to be one of the key features that distinguishes HOS from other systems theories, such as that of Mesarovic and Takahara (1975). Mesarovic and Takahara define a system as a relation, i.e., a particular kind of set; "...to enable a system or its restrictions, which are both in general relations, to be represented as functions...new auxiliary objects, termed state objects, had to be introduced" (p. 10). This situation seems to bother Mesarovic and Takahara somewhat, and they make some attempt to justify it, arguing that it is necessary because of the nature of mathematics.

> A system is defined as a set (of a particular kind, i.e., a relation). It stands for the collection of all appearances of the object of study rather than for the object of study itself. This is necessitated by the use of mathematics as the language for the theory in which a 'mechanism' (a function or a relation) is defined as a set, i.e., as a collection of all proper combinations of components. Such a characterization of a system ought not to create any difficulty since the set relation, with additional specifications, contains all the information about the actual 'mechanism' we can legitimately use in the development of a formal theory (p. 7).

As we have seen here, however, it really is not necessary at all in mathematics to define a function as a set, since functions can just as well be treated as unitary basic objects, as in category theory.

Mesarovic and Takahara's apparent discomfort with the treatment of systems as sets seems to reflect a genuine intuition about systems that is captured directly in the HOS approach. Real-world systems are things that do things,

not just collections of objects or their n-tuples. While we can develop coherent, even useful, theories that treat systems as sets, introducing their functional character--the processes they undergo--only indirectly after the fact, it would seem intuitively more natural and more in accord with the actual character of real-world systems to recognize a system's function as its primary aspect and to base our theory on that recognition. This is essentially the approach that is taken in HOS.

REFERENCES

Arbib, M. A. and E. G. Manes, Arrows, Structures, and Functors. Academic Press: New York, 1975.

Cushing, S., "The Software Security Problem and How to Solve It," TR-6, Revision 1. Higher Order Software, Inc.: Cambridge, MA, July 1977.

Cushing, S., "Algebraic Specification of Data Types in Higher Order Software (HOS)," Proceedings, Eleventh Hawaii International Conference on System Sciences: Honolulu, Hawaii, January 1978.

Hamilton, M. and S. Zeldin, "Higher Order Software Techniques Applied to a Space Shuttle Prototype Program," Lecture Notes in Computer Science, Vol. 19, Goos and J. Harmanis, Eds., Springer-Verlag: New York, pp. 17-31. Presented at Program Symp. Proc. Colloque sur la Programmation, Paris, France, August 1973.

Hamilton, M. and S. Zeldin, "The Foundations for AXES: A Specification Language Based on Completeness of Control, Doc. R-964. Charles Stark Draper Laboratory, Inc.: Cambridge, MA, March 1976(a).

Hamilton, M. and S. Zeldin, "Higher Order Software--A Methodology for Defining Software," IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976(b).

Mesarovic, M. D. and Y. Takahara, General Systems Theory: Mathematical Foundations. Academic Press: New York, 1975.

Whitehead, A. N., Science and the Modern World. MacMillan Co.: Riverside, NJ, 1967.

Part 8

HOW TO DO A DATA TYPE

S. Cushing

(1) LIST ALL OPERATIONS YOU THINK MIGHT BE USEFUL IN CONNECTION WITH THE TYPE:

OPERATIONS ON THE TYPE MEMBERS

AND

OPERATIONS THAT PRODUCE TYPE MEMBERS

(2) LIST ALL PROPERTIES YOU THINK MEMBERS OF THE TYPE MUST HAVE; LOOK OUT ESPECIALLY FOR INVERSE OPERATIONS AND DISTINGUISHED ELEMENTS;
ALL THESE PROPERTIES MUST BE EXPRESSED IN AXIOM FORM IN TERMS OF OPERATIONS IN (1) OR OPERATIONS ALREADY AVAILABLE ON OTHER TYPES;
IF NECESSARY, ADD NEW OPERATIONS TO (1) TO GET (1'), AND RECORD ANY DISTINGUISHED ELEMENTS.

(3) DETERMINE FROM (2) WHICH OPERATIONS IN (1') CAN BE EXPRESSED EXPLICITLY IN TERMS OF OTHERS: REMOVE THOSE OPERATIONS FROM (1') TO GET (1"); REMOVE THE RELEVANT PROPERTIES FROM (2) TO GET (2')

(THERE MAY BE ALTERNATE CHOICES POSSIBLE; USE YOUR JUDGEMENT AND REMEMBER WHAT CHOICES WERE MADE, IN CASE YOU LATER CHANGE YOUR MIND)

(4) DETERMINE WHICH PROPERTIES IN (2') CAN BE PROVEN FROM OTHERS AND REMOVE THEM FROM (2') TO GET (2").

(AGAIN, THERE MAY BE CHOICES AND THE SAME PROVISO HOLDS)

THIS GIVES YOUR TENTATIVE AXIOM SET (TAS)

(5) TRY VERY HARD TO DERIVE A CONTRADICTION FROM
THE AXIOMS IN TAS: IF YOU CAN, THE AXIOMS ARE
INCONSISTENT;
DETERMINE WHERE THE INCONSISTENCY RESIDES AND
MODIFY TAS ACCORDINGLY;
THEN REDO THIS STEP AS NECESSARY:
WHEN YOU CAN NO LONGER DERIVE A CONTRADICTION,
YOU ARE AT LEAST ON THE RIGHT TRACK;
THE AXIOMS MAY BE CONSISTENT.

(6) CONSTRUCT A MODEL FOR THE AXIOMS OR SHOW THAT THEY INSTANTIATE A MORE GENERAL AXIOM SYSTEM KNOWN TO BE CONSISTENT;
IF EITHER IS POSSIBLE, THE AXIOMS ARE CONSISTENT.

NOTE 1: CONSISTENCY

A PHYSICAL MODEL FOR THE AXIOMS PROVES CONSISTENCY ABSOLUTELY: IF SOMETHING EXISTS, ITS PROPERTIES ARE CONSISTENT.

A MATHEMATICAL MODEL OR INSTANTIATION PROOF PROVES CONSISTENCY RELATIVE TO THE CONSISTENCY OF THE MODEL OR MORE GENERAL SYSTEM: FOR PRACTICAL PURPOSES, THIS IS ENOUGH, BECAUSE IF MATHEMATICS ITSELF TURNS OUT TO BE INCONSISTENT, ALL BETS ARE OFF.

NOTE 2: COMPLETENESS

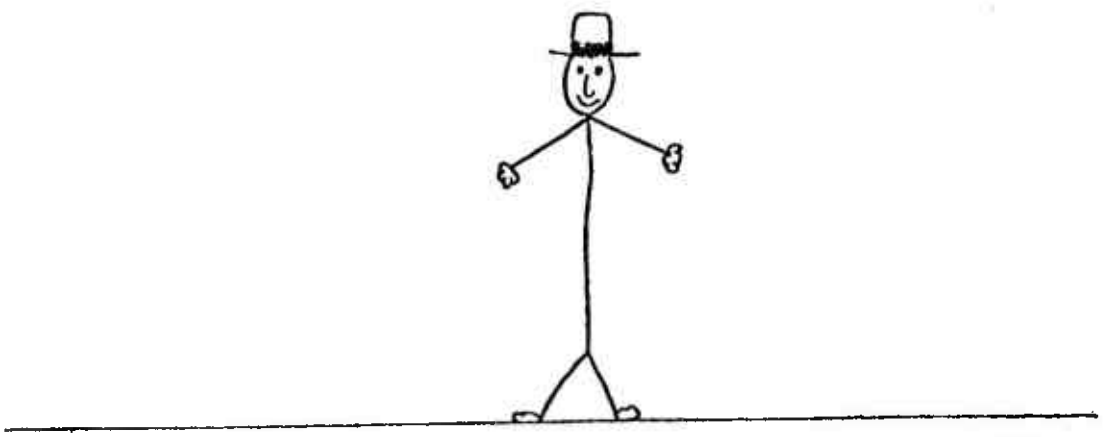
THERE IS A FUNDAMENTAL UPPER BOUND ON COMPLETENESS OF TYPES:

GODEL'S THEOREM: ANY CONSISTENT AXIOMATIC SYSTEM THAT IS POWERFUL ENOUGH TO INCLUDE ARITHMETIC IS INCOMPLETE.

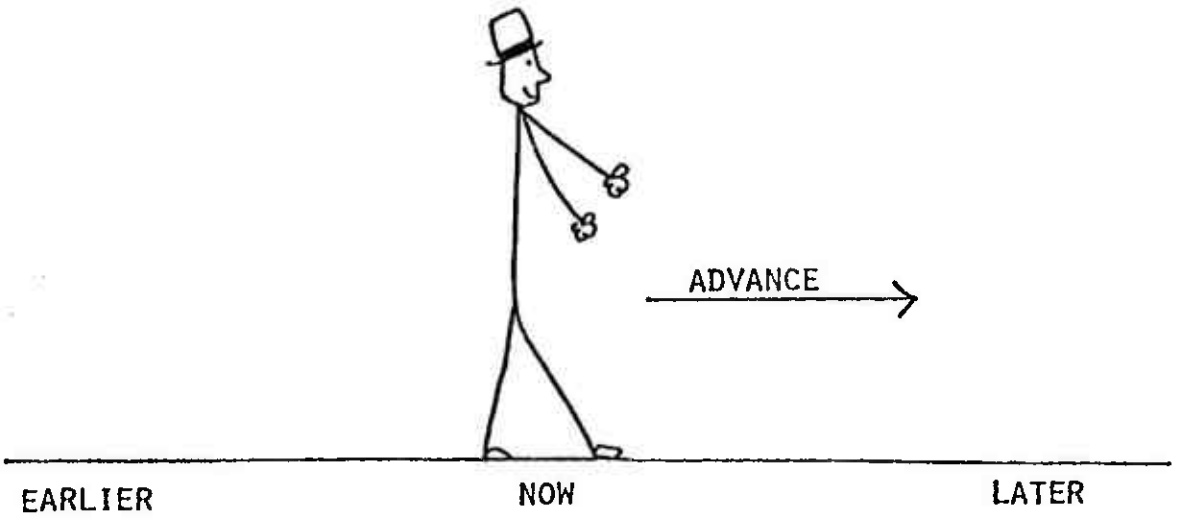
I.E., NO MATTER HOW MANY AXIOMS WE HAVE FOR A TYPE THAT RELATES TO THE NATURAL NUMBERS, SOME PROPERTIES OF ITS MEMBERS WILL NOT BE PROVABLE FROM THE AXIOMS: THESE PROPERTIES ARE USUALLY VERY OBSCURE, HOWEVER; WE CAN ALWAYS AXIOMATIZE COMPLETELY FOR A GIVEN SET OF PROPERTIES, WHICH IT'S UP TO US TO SPECIFY; THUS COMPLETENESS IS ALWAYS RELATIVE TO WHAT YOU HAVE IN MIND; WHAT PROPERTIES DO YOU WANT THE SYSTEM TO HAVE?

EXAMPLE: TIME

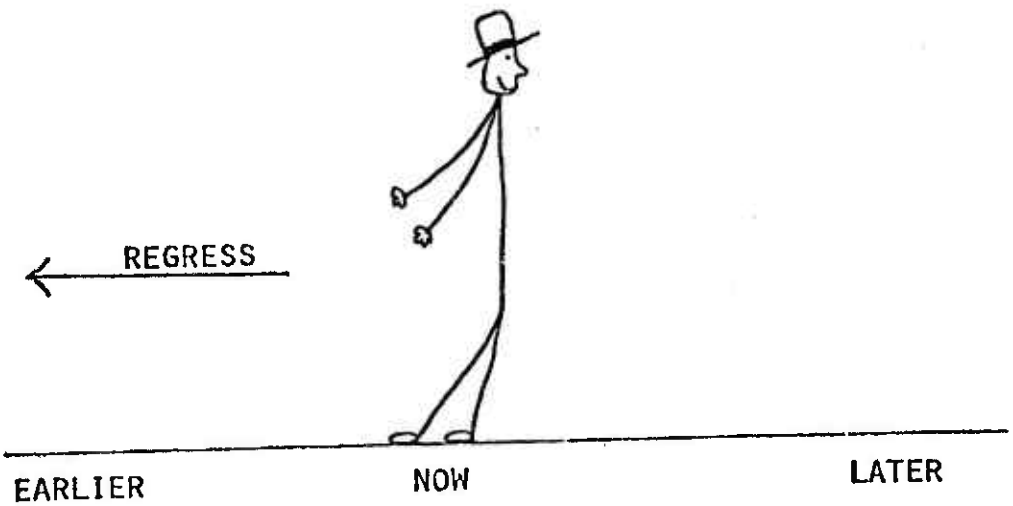
YOU MAY MEAN BY "TIME" JUST A LINEARLY ORDERED SEQUENCE: IN THAT CASE, ALL YOU NEED IS A LINEAR ORDERING.



YOU MAY WANT TO TALK ABOUT TIME GOING FORWARD, BUT NEVER HAVE IT GO BACKWARD: IN THAT CASE, YOU ALSO NEED AN OPERATION ADVANCE



YOU MAY WANT TO BE ABLE TO RUN YOUR SYSTEM BACKWARD:
IN THAT CASE, YOU ALSO NEED AN OPERATION REGRESS



BUT WATCH OUT: WE WON'T NEED REGRESS AS A PRIMITIVE OPERATION, IF WE HAVE ADVANCE AND REVERSE, BECAUSE IT CAN BE EXPRESSED IN TERMS OF THEM AS A CONTROL MAP

EXAMPLE:

HOW TO DO TIME

(1) LIST ALL OPERATIONS YOU THINK MIGHT BE USEFUL ON THE TYPE:

Before: operates on times, produces booleans

After: operates on times, produces booleans

Notafter: operates on times, produces booleans

Notbefore: operates on times, produces booleans

Advance: operates on times, produces times

Reverse: operates on times, produces times

Regress: operates on times, produces times

(2) LIST ALL PROPERTIES YOU THINK MEMBERS OF THE TYPE MUST HAVE:

properties of Before: Before(t,t) = False etc.

properties of After: After(t,t) = False

((After(t_1,t_2) & After(t_2,t_3)) ⊃ After(t_1,t_3)) = True etc.

properties of Notafter: Notafter(t,t) = True

(Notafter(t_1,t_2) & Notafter(t_2,t_1)) ⊃ Equal(t_1,t_2)) = True

Notafter(t_1,t_2) ! Notafter(t_2,t_1) = True etc.

properties of Notbefore: similar to Notafter

properties of Advance: Advance(t,Notime) = t
(Note that Notime must be recorded as a distinguished element)

Advance(t_1,t_2) = Advance(t_2,t_1)

Advance(t_1,Advance(t_2,t_3)) = Advance(Advance(t_1,t_2),t_3) etc.

mixed properties: Advance(Reverse(t),t) = Notime

Notafter(Advance(t_1,t_2),t_1) = Notafter(t_2,Notime)

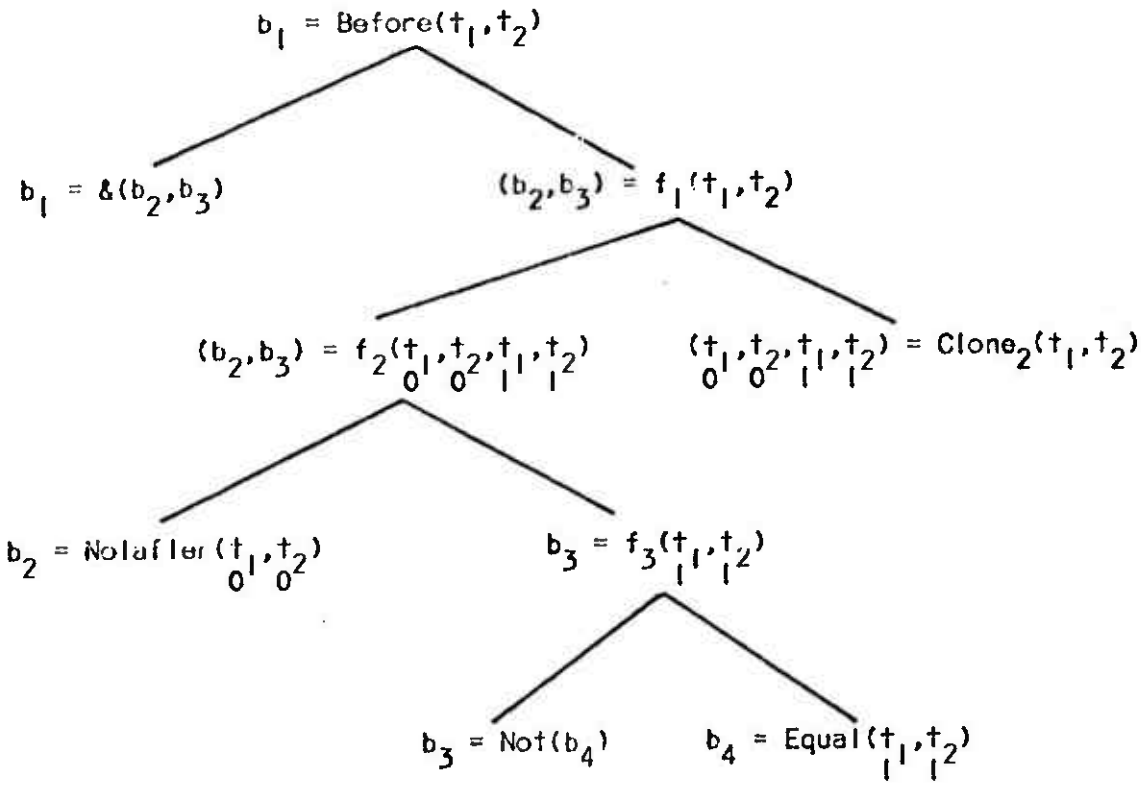
Advance(Regress(t_1,t_2),t_2) = t_1 etc.

(3) DETERMINE FROM (2) WHICH OPERATIONS IN (1') CAN BE EXPRESSED IN TERMS OF OTHERS:

AS AN EQUATION:

$$Before(t_1,t_2) = Notafter(t_1,t_2) \;\&\; Not(Equal(t_1,t_2))$$

OR AS A CONTROL MAP:



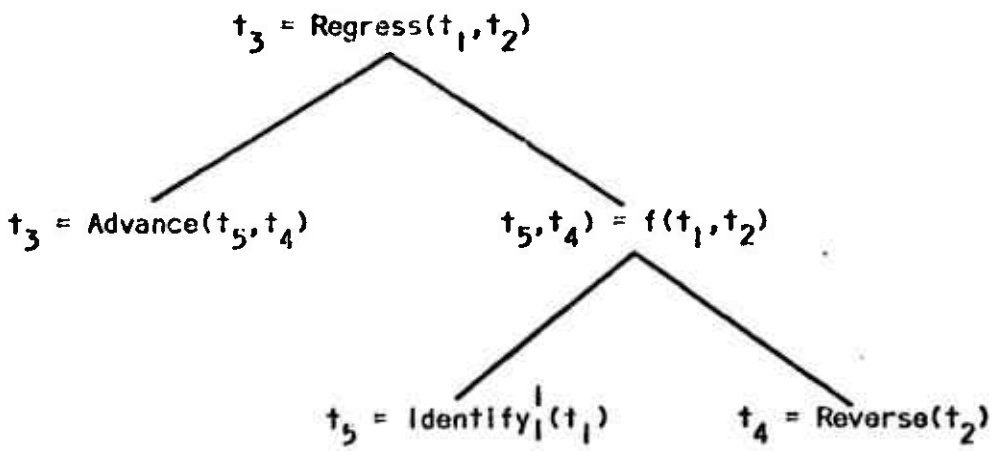
SO WE CAN REMOVE BEFORE FROM (1'), KEEPING NOTAFTER

SIMILARLY,

AS AN EQUATION:

$$Regress(t_1, t_2) = Advance(t_1, Reverse(t_2))$$

OR AS A CONTROL MAP:

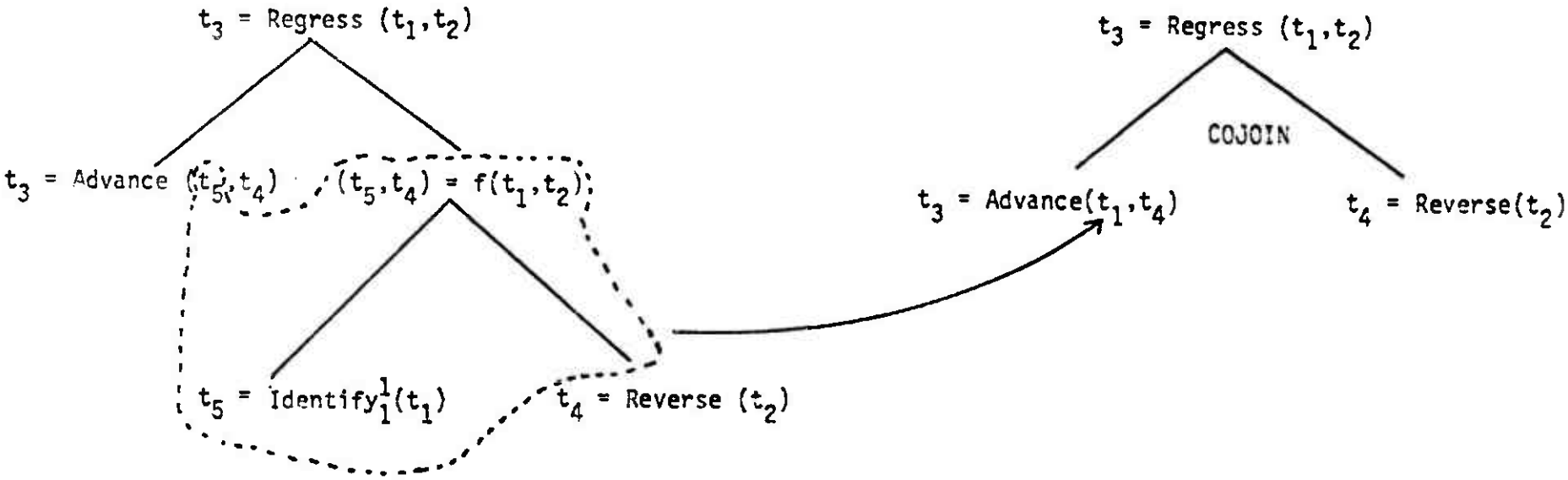


SO WE CAN DELETE REGRESS FROM (1'), KEEPING ADVANCE AND REVERSE

ETC.

NOTE: THESE PARTICULAR CONTROL MAPS ARE MORE COMPLICATED THAN THEY NEED BE, BECAUSE ONLY PRIMITIVE CONTROL STRUCTURES ARE USED.
THIS IS FOR EXPOSITORY PURPOSES ONLY.
CONTROL MAPS CAN BE GREATLY SIMPLIFIED BY USING ABSTRACT CONTROL STRUCTURES -- AND REMEMBER:

IT GETS EASIER WITH PRACTICE



(a) Decomposition in Terms of Primitive Control Structures

(b) Decomposition in terms of the Abstract Control Structure COJOIN

Two Control Maps for Operation Regress on Data Type TIME

(4) DETERMINE WHICH PROPERTIES IN (2') CAN BE PROVEN FROM OTHERS:

SUPPOSE WE HAVE THE THREE AXIOMS

(1) $\text{Notafter}(t,t) = \text{True}$

(2) $((\text{Notafter}(t_1,t_2) \;\&\; \text{Notafter}(t_2,t_1)) \supset \text{Equal}(t_1,t_2)) = \text{True}$

(3) $\text{Notafter}(t_1,t_2) \;\&\; \text{Notafter}(t_2,t_1) = \text{Equal}(t_1,t_2)$

HERE (3) CAN BE PROVEN FROM (2) AND (1), AND EACH OF (2) AND (1) CAN BE PROVEN FROM (3), SO WE DON'T NEED ALL THREE AXIOMS.

USE YOUR JUDGEMENT.

KEEP EITHER (3) OR (2) AND (1), DEPENDING ON WHAT YOUR EXPERIENCE AND YOUR KNOWLEDGE OF YOUR PARTICULAR PROBLEM TELL YOU WILL BE MOST USEFUL.

STEPS (1) - (4) GIVE US A TENTATIVE AXIOM SET SOMETHING LIKE THE FOLLOWING (DEPENDING ON THE CHOICES THAT WERE MADE):

(1) Notafter(t,t) = True

(2) ((Notafter(t_1,t_2) & Notafter(t_2,t_3)) ⊃ Notafter(t_1,t_3)) = True

(3) ((Notafter(t_1,t_2) & Notafter(t_2,t_1)) ⊃ Equal(t_1,t_2)) = True

(4) Notafter(t_1,t_2) ! Notafter(t_2,t_1) = True

(5) Advance(t,Notime) = t

(6) Advance(t_1,t_2) = Advance(t_2,t_1)

(7) Advance(t_1,Advance(t_2,t_3)) = Advance(Advance(t_1,t_2),t_3)

(8) Notafter(Advance(t_1,t_2),t_1) = Notafter(t_2,Notime)

(9) Advance(Reverse(t),t) = Notime

(5) TRY TO DERIVE A CONTRADICTION: IN THIS CASE YOU CAN'T (TRY IT!)

BUT SUPPOSE WE HAD GOOFED AND WRITTEN

(*) Notafter(t,t) = False

INSTEAD OF AXIOM 1; THEN LET

$t_1 = t$

$t_2 = t$

THIS GIVES

$$\text{Notafter}(t_1,t_2) = \text{Notafter}(t,t) = \text{False}$$

and $$\text{Notafter}(t_2,t_1) = \text{Notafter}(t,t) = \text{False}$$

SO WE GET

$$\text{Notafter}(t_1,t_2) \mid \text{Notafter}(t_2,t_1) = \text{False}$$

BUT THIS CONTRADICTS AXIOM 4!

THIS WOULD REQUIRE RE-EXAMINING THE AXIOMS TO SEE WHAT WENT WRONG; WITH A LITTLE THOUGHT WE WOULD FIGURE OUT THAT (*) SHOULD HAVE BEEN 1.

(6) CONSTRUCT A MODEL FOR THE AXIOMS:
IN THIS CASE A MATHEMATICAL MODEL CAN BE CONSTRUCTED

FOR TIME TAKE THE RATIONAL NUMBERS

Notafter	$\leq$
Advance	+
Reverse	0-
Notime	0

THE AXIOMS THEN BECOME

(1) $(r \leq r) = \text{True}$ TRUE!

(2) $((r_1 \leq r_2 \ \& \ r_2 \leq r_3) \supset r_1 \leq r_3) = \text{True}$ TRUE!

(3) $((r_1 \leq r_2 \ \& \ r_2 \leq r_1) \supset r_1 = r_2) = \text{True}$ TRUE!

(4) $(r_1 \leq r_2 \ ! \ r_2 \leq r_1) = \text{True}$ TRUE!

(5) $r + 0 = r$ TRUE!

(6) $r_1 + r_2 = r_2 + r_1$ TRUE!

(7) $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$ TRUE!

(8) $((r_1 + r_2) \leq r_1) = (r_2 \leq 0)$ TRUE!

(9) $(-t) + t = 0$ TRUE!

SINCE THE AXIOMS HAVE A MODEL THEY ARE CONSISTENT

NOTE: THE RATIONALS ARE A MODEL FOR THESE TIME AXIOMS
BUT NOT THE ONLY MODEL;
ONCE WE HAVE A MODEL, WE KNOW THE AXIOMS ARE CONSISTENT;
BUT ONCE WE KNOW THE AXIOMS ARE CONSISTENT,
WE CAN THEN CHOOSE ANY MODEL;
IN OTHER WORDS, ANY IMPLEMENTATION OF THE OPERATIONS
IS OKAY, AS LONG AS IT SATISFIES THE AXIOMS;
WHICH IMPLEMENTATION YOU CHOOSE DEPENDS ON YOUR
APPLICATION AND YOUR JUDGEMENT

DATA TYPE: TIME;

PRIMITIVE OPERATIONS:

$time_3$ = Advance($time_1$,$time_2$);

boolean = Notafter($time_1$,$time_2$);

$time_2$ = Reverse($time_1$);

AXIOMS:

WHERE t,t_1,t_2,t_3 ARE TIMES;

WHERE Notime IS A CONSTANT TIME;

Notafter(t,t) = True; (1)

((Notafter(t_1,t_2) & Notafter(t_2,t_3)) ⊃ Notafter(t_1,t_3)) = True; (2)

((Notafter(t_1,t_2) & Notafter(t_2,t_1)) ⊃ Equal(t_1,t_2)) = True; (3)

Notafter(t_1,t_2) ! Notafter(t_2,t_1) = True; (4)

Advance(t,Notime) = t; (5)

Advance(t_1,t_2) = Advance(t_2,t_1); (6)

Advance(t_1,Advance(t_2,t_3)) = Advance(Advance(t_1,t_2),t_3); (7)

Notafter(Advance(t_1,t_2),t_1) = Notafter(t_2,Notime); (8)

Advance(Reverse(t),t) = Notime; (9)

END TIME;

Group = $[\Sigma, \omega]$

Σ: G, Neut $\in$ G

ω: Mult: G x G $\rightarrow$ G

Inv: G $\rightarrow$ G

Axioms: For all $g, g_1, g_2, g_3 \in G$,

1. Mult $(g_1, \text{Mult}(g_2, g_3)) = \text{Mult}(\text{Mult}(g_1, g_2), g_3)$
2. Mult (Neut, g) = g
3. Mult (g, Neut) = g
4. Mult (g, Inv (g)) = Neut

The Groups as Algebras

Σ: INTEGERS, ZERO $\in$ INTEGERS

ω: +: INTEGERS x INTEGERS $\rightarrow$ INTEGERS

-: INTEGERS $\rightarrow$ INTEGERS

Axioms: For all $i, i_1, i_2, i_3 \in$ INTEGERS

1. $(i_1 + (i_2 + i_3)) = ((i_1 + i_2) + i_3)$
2. Zero + i = i
3. i + Zero = i
4. i + (-i) = Zero

The Integers as a Group

Σ: PRATIONALS, One $\in$ PRATIONALS

ω: $\cdot$: PRATIONALS x PRATIONALS $\rightarrow$ PRATIONALS

$^1/$: PRATIONALS $\rightarrow$ PRATIONALS

Axioms: For $p, p_1, p_2, p_3 \in$ PRATIONALS

1. $(p_1 \cdot (p_2 \cdot p_3)) = ((p_1 \cdot p_2) \cdot p_3)$
2. One $\cdot$ p = p
3. p $\cdot$ One = p
4. p $\cdot$ (1/p) = One

The Positive Rationals as a Group

DATA TYPE SCALAR

DATA TYPE: SCALAR;

PRIMITIVE OPERATIONS:

$scalar_3$ = SSum($scalar_1$,$scalar_2$);

$scalar_3$ = SMult($scalar_1$,$scalar_2$);

$scalar_2$ = SOpp($scalar_1$);

$scalar_2$ = SInv($scalar_1$);

AXIOMS:

WHERE sc,sc_1,sc_2,sc_3 ARE SCALARS;

WHERE SZero,SUnit ARE CONSTANT SCALARS;

SSum(sc_1,SSum(sc_2,sc_3)) = SSum(SSum(sc_1,sc_2),sc_3); (1)

SSum(sc_1,sc_2) = SSum(sc_2,sc_1); (2)

SSum(sc,SZero) = sc; (3)

SSum(sc,SOpp(sc)) = SZero; (4)

SMult(sc_1,SMult(sc_2,sc_3)) = SMult(SMult(sc_1,sc_2),sc_3); (5)

SMult(sc_1,sc_2) = SMult(sc_2,sc_1); (6)

SMult(sc,SUnit) = sc; (7)

SInv(SZero) = REJECT; (8)

SMult(sc,SInv(sc)) = K_{Unit}(^{1}sc) AND K_{REJECT}(^{2}sc); (9)

PARTITION OF sc IS

^{1}sc|sc ≠ Szero,

^{2}sc|sc = SZero;

SMult(sc_1,SSum(sc_2,sc_3)) = SSum(SMult(sc_1,sc_2),SMult(sc_1,sc_3)); (10)

DATA TYPE VECTOR

DATA TYPE: VECTOR;

PRIMITIVE OPERATIONS:

$vector_3$ = VSum($vector_1$,$vector_2$);

$vector_2$ = VOpp($vector_1$);

$vector_2$ = VMult(scalar,$vector_1$);

AXIOMS:

WHERE v,v_1,v_2,v_3 ARE VECTORS;

WHERE sc,sc_1,sc_2 ARE VECTORS;

WHERE VZero IS A CONSTANT VECTOR;

VSum(v_1,VSum(v_2,v_3)) = VSum(VSum(v_1,v_2),v_3); (1)

VSum(v_1,v_2) = VSum(v_2,v_1); (2)

VSum(v,VZero) = v; (3)

VSum(v,VOpp(v)) = VZero; (4)

VMult(SUnit,v) = v; (5)

VMult(SSum(sc_1,sc_2),v) = VSum(VMult(sc_1,v),VMult(sc_2,v)); (6)

VMult(sc,VSum(v_1,v_2)) = VSum(VMult(sc,v_1),VMult(sc,v_2)); (7)

VMult(SMult(sc_1,sc_2),v) = VMult(sc_1,VMult(sc_2,v)); (8)

END VECTOR;

DATA TYPE ADDRESS

DATA TYPE: ADDRESS;

PRIMITIVE OPERATIONS:

$boolean_2$ = Equal($address_1$,$address_2$);

AXIOMS:

WHERE A_1,A_2 ARE ADDRESSES;

Equal(A_1,A_1) = True;

Equal(A_1,A_2) = Equal(A_2,A_1);

Entails(Equal(A_1,A_2) & Equal(A_2,A_3),Equal(A_1,A_3)) = True;

END ADDRESS;

DATA TYPE: ADDRESS;

PRIMITIVE OPERATIONS:

AXIOMS:

END ADDRESS;

DATA TYPE STACK

DATA TYPE: STACK;

PRIMITIVE OPERATIONS:

$stack_1$ = Push($stack_2$, $integer_1$);

$stack_1$ = Pop($stack_2$);

$Integer_1$ = Top($stack_1$);

AXIOMS:

WHERE Newstack IS A CONSTANT STACK;

WHERE s IS A STACK;

WHERE i IS AN INTEGER;

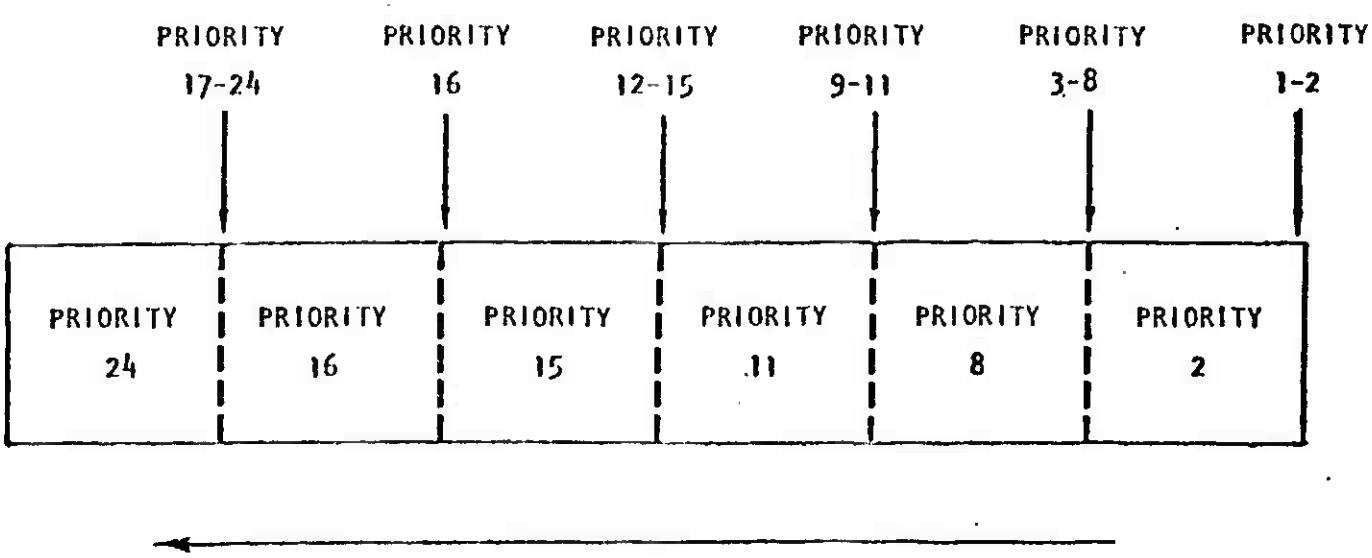
Top(Newstack) = REJECT;

Top(Push(s,i)) = i;

Pop(Newstack) = REJECT

Pop(Push(s,i)) = s;

END STACK;



STRICT PRIORITY QUEUE: Entrance strictly by priority; arrows show exit point, entrance points, and direction of flow through queue.

DATA TYPE STRICT PRIORITY QUEUE

DATA TYPE: QUEUE;

PRIMITIVE OPERATIONS;

$queue_2$ = Add(job,$queue_1$);

$queue_2$ = Remove($queue_1$);

job = Front(queue);

boolean = Equal($queue_1$,$queue_2$);

natural = Size(queue);

AXIOMS:
WHERE Nullq IS A CONSTANT QUEUE,

WHERE Capacity IS A CONSTANT NATURAL;

1. Front(Nullq) = REJECT;
2. Front(Add(j,q)) = ^{1}j AND Front(^{2}q) AND K_{REJECT}(^{3}q);
3. Remove(Nullq) = REJECT;
4. Remove(Add(j,q)) = ^{1}q AND Add(^{2}j,Remove(^{2}q)) AND K_{REJECT}(^{3}q);

PARTITION OF (j,q) IS

^{1}(j,q) | (Equal(q,Nullq) !> (Priority(j),Priority(Front(q)))) &
Size(q)<Capacity,

^{2}(j,q) | (Not(Equal(q,Nullq))&≤(Priority(j),Priority(Front(q))))
& Size(q)<Capacity,

^{3}(j,q) | Size(q)≥Capacity;

5. Equal(Nullq,Nullq) = True;
6. Equal(Nullq,Add(j,q)) = False;
7. Equal(Add(j,q),Nullq) = False;
8. Equal(Add(j_1,q_1), Add(j_2,q_2)) = Equal(j_1,j_2)& Equal(q_1,q_2);
9. Size(Nullq) = Zero;
10. Size(Add(j,q)) = Succ(Size(^{1}q)) AND Succ(Size(^{2}q))AND K_{REJECT}(^{3}q);

END QUEUE;

Part 9

FUZZY SETS AND APPROXIMATE REASONING

L. Vaina

"Both precision and certainty are false ideals. They are impossible to attain, and therefore dangerously misleading if they are uncritically accepted as guides. The quest for precision is analogous to the quest for certainty and both should be abandoned. I do not suggest, of course, that an increase in the precision of, say, a prediction, or even a formulation, may not sometimes be highly desirable. What I do suggest is that it is always undesirable to make an effort to increase precision for its own sake--especially linguistic precision--since this usually leads to lack of clarity, and to a waste of time and effort... One should never try to be more precise than the problem situation demands."

- Karl Popper
Unended Quest, 1976

FUZZY SETS AND APPROXIMATE REASONING

Much of the decision making in the real world takes place in an environment in which the goals, constraints, and consequences of possible actions are not known precisely. To deal quantitatively with this imprecision, we usually employ the concepts and techniques of probability theory, and, more particularly, the tools provided by decision theory, control theory, and information theory. In doing so, we are tacitly accepting the premise that imprecision, whatever its nature, can be equated with randomness. This, in my view, is a questionable assumption. I claim that there is a need for differentiation between randomness and fuzziness, with the latter being a major source of imprecision in many decision processes. By fuzziness we mean a type of imprecision which is associated with fuzzy sets; that is, classes where there is no sharp transition from membership to non-membership. In sharp contrast to the notion of a class or a set in mathematics, most of the classes in the real world do not have crisp boundaries which separate those objects which belong to a class from those which do not.

What is the distinction between randomness and fuzziness? Randomness has to do with uncertainty concerning membership or non-membership of an object in a nonfuzzy set. Fuzziness has to do with classes in which there may be grades of membership intermediate between full membership and non-membership.

> Example: "The grade of membership of John to the class of tall men is 0.7 " is a nonprobabilistic statement concerning the membership of John in the fuzzy class of tall men, whereas "the probability that John will get married within a year is 0.7" is a probabilistic statement concerning the uncertainty of the occurrence of a nonfuzzy event (marriage).

Reflecting this distinction, the mathematical techniques for dealing with fuzziness are quite different from those of probability theory. They are simpler in many ways because to the notion of probability measure in probability theory corresponds the simpler notion of membership function in the theory of fuzziness.

In speaking of the variety of imprecision, a point that is in need of clarification relates to the distinction between fuzziness and vagueness. Vagueness is viewed as a particular form of fuzziness. So, a fuzzy proposition, "John is quite tall" is fuzzy by virtue of the fuzziness of the class quite tall. A vague proposition, on the other hand, is one which is (1) fuzzy and (2) ambiguous--in the sense of providing insufficient information for a particular purpose. So, "John is quite tall" may not be sufficiently specific for deciding which size of car to buy for John. In this case, the propositional question is both fuzzy and ambiguous, and hence is vague. On the other hand, "John is quite tall" may provide sufficient information for choosing a tie for John, in which case the proposition in question is fuzzy but not vague. Vagueness is an application context-dependent characteristic of a proposition, whereas fuzzy is not.

The theory of fuzzy sets has two distinct branches. In one, a fuzzy set is treated as a mathematical construct concerning which one can make provable assertions. This "nonfuzzy" theory of fuzzy sets is in the spirit of traditional mathematics and is typified by the rapidly growing literature on fuzzy topological spaces, fuzzy switching function, fuzzy orderings, etc. The other branch may be viewed as a "fuzzy" theory of fuzzy sets in which fuzziness is introduced into the logic which underlies the rules of manipulation of fuzzy sets and assertions about them. The genesis of this branch of the theory is related to the introduction of the so-called linguistic approach, which in turn, has led to the development of fuzzy logic. In this logic the truth-value as well as the rules inference are allowed to be imprecise with the result that the assertions about fuzzy sets based on this logic are not, in general, provable propositions in two-valued logic. For example, the proposition "John is very intelligent" may be "more or less true." The fuzzy "theory" of fuzzy sets is still in its initial stages of development, but it is important as a foundation for approximate reasoning, or equivalently, fuzzy reasoning. Such reasoning characterizes much of the human thinking and is the basis of the remarkable human ability to attain inprecise specified goals in an incompletely unknown environment.

Zadeh

> ... in general complexity and precision bear an inverse relation to one another in the sense that, as the complexity of a problem increases, the possibility of analyzing it in precise terms diminishes. Thus, 'fuzzy thinking' may not be deplorable, after all, if it makes possible the solution of problems which are much too complex for precise analysis.

The essence and power of human reasoning is in its capability to grasp and use inexact concepts directly. Zadeh argues that attempts to model or emulate it by formal systems of increasing precision will lead to decreasing validity and relevance.

However, it is important to keep in mind that Zadeh's analysis of human reasoning processes and his exposition of fuzzy set theory are not one and the same--they must be separated conceptually. Fuzzy sets are to approximate reasoning what lattice theory is to a propositional calculus: a vital mathematical tool for certain approaches to the theory, but not the theory itself.

Fuzzy Sets

Informally, we have seen that a fuzzy set is a class of objects in which there is no sharp boundary between those objects that belong to the class and those that do not.

> Definition: Let $X = \{x\}$ denote a collection of objects X. A fuzzy set A in X is a set of ordered pairs $A = \{(x, \Gamma_A(X))\}$, $x \in X$, where $\Gamma_A(x)$ is termed the grade of membership of x in A, and $\Gamma_A: X \to M$ is a function from X to space M called the membership space. When $M = \{0,1\}$, A is nonfuzzy and its membership function becomes identical with the characteristic function of nonfuzzy set. In general, we assume that $M = [0,1]$. So, the fuzzy set A, despite the unsharpness of its boundaries, can be defined precisely by associating with each object X a number between 0 and 1 which represents its grade of membership in A.

NOTATION: $A = \{x|x \cong 5\}$ will denote the set of numbers which are approximately equal to 5. The symbol ~ will be referred to as a fuzzifier.

NORMALITY: A fuzzy set A is normal iff $\text{Sup}_x \Gamma_A(x) = 1$, the supranum of $\Gamma_a(x)$ over X is unity. A fuzzy set is subnormal if it is not normal.

SUPPORT: The support of a fuzzy set A is a set S(A) such that $X \in S(A) \Longleftrightarrow \Gamma_A(x) > 0$. If $\Gamma_A(x)$ = constant over S(A), A is nonfuzzy.

EQUALITY: Two fuzzy sets are equal, A=B, iff $\Gamma_A(x) = \Gamma_B(x)$,

CONTAINMENT: A fuzzy set A is a subset of a fuzzy set B, $A \subset B$, if $\Gamma_A(x) < \Gamma_B(x)$,

COMPLEMENTATION: A' is the complement of A iff $\Gamma_{A'}(x) = 1-\Gamma_A(x)$,

INTERSECTION: $A \cap B$ is the largest fuzzy set that contains both A and B; $\Gamma_{A \cap B}(x) = \text{Min}(\Gamma_A(x), \Gamma_B(x))$, $(\forall)\ x \in X$

UNION: $A \cup B$, $\Gamma_{A \cup B}(x) = \text{Max}\ (\Gamma_A(x), \Gamma_B(x))$ $(\forall)\ x \in X$

Example: Fuzzy Goals, Constraints, and Decisions

In the conventional approach to decision making, the principal ingredients of a decision process are:

(a) a set of alternatives
(b) a set of constraints on the choice between different alternatives
(c) a performance function which associates with each alternative the gain resulting from the choice of that alternative.

When we view a decision process from the broader perspective of decision making in a fuzzy environment, a different and perhaps more natural conceptual framework suggests itself. The most important feature of this framework is its symmetry with respect to goals and constraints--a symmetry which erases the differences between them and makes its possible to relate to a relatively simple way the concept of decision to those of the goals and constraints of a decision process.

Let X = {x} be a given set of alternatives. A fuzzy goal G, in x is identified with a given fuzzy set G in X.

Example: $X = R^1$ (the real line), then the fuzzy goal expressed in words "x should be substantially larger than 10," might be represented by a fuzzy set in R^1 whose membership function is (subjectively) given by:

$$(A)\quad \Gamma_G(x) = \begin{cases} 0 & x < 10 \\ (1 + (x-10)^{-2})^{-1} & x \geq 10 \end{cases}$$

A Fuzzy Constraint: C, in X, is defined by a fuzzy set in X.

Example: In R^1, "x should be approximative between 2 and 10," could be represented by the fuzzy sets with the membership function.

(B) $\Gamma_C(x) = (1 + a(x-6)^m)^{-2}$, where a is a positive number, m is a positive even integer, chosen in such a way as to reflect the sense in which the approximation of the interval [2,10] is to be understood. For example, if we set m=4, $a=5^{-4}$, then at x=2 and x=10 we have $\Gamma_C(x) = \sim 0.71$. The constraints and goals being defined as fuzzy sets in the space of alternatives, thus can be treated identically in the formulation of a decision.

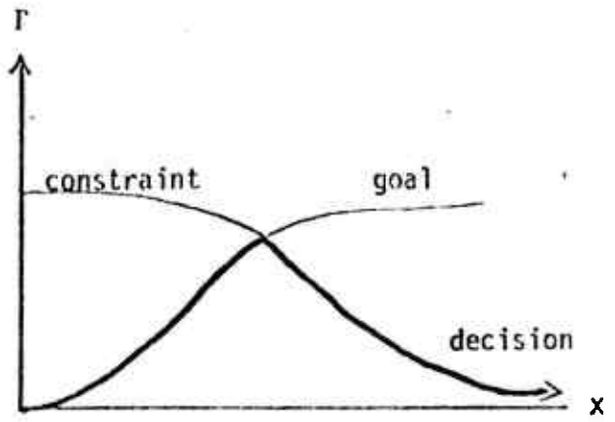
Example: G: x should be substantially larger than 10,
C: x should be in the vicinity of 15, or $\Gamma_G(x)$ given by (A) and $\Gamma_C(x)$ by (B).

DECISION: as a choice, or a set of choices drawn from the available alternatives; we define a fuzzy decision as the fuzzy set of alternatives resulting from the intersection of the goals and constraints.

$$\Gamma_D(x) = \Gamma_{G \cap C}(x) = \Gamma_G(x) \wedge \Gamma_C(x), \quad (\forall) x \in X$$

$$\Gamma_{G \cap C}(x) = \begin{cases} \text{Min}((1 + (x-10)^{-2})^{-1}, (1 + (x-15)^4)^{-1} & x \geq 10 \\ 0 & x < 10 \end{cases}$$

The relation between G, C, D is depicted in



Let K be the set of points in x on which Γ_D attains its maximum. Then the nonfuzzy subset D^M of D defined by

$$\Gamma_{D^M}(x) = \begin{cases} \text{Max } \Gamma_D(x), & x \in X \\ 0 \text{ elsewhere} \end{cases}$$

will be said to be the optimal decision.

In defining a fuzzy decision D as the intersection of the goals and constraints, we assume that all of the goals and constraints are of equal importance. There are more cases where some of the goals or some of the constraints are of the greater importance than others. In such cases D might be expressed as a convex combination of the goals and constraints, with the weighing coefficients relfecting the relative importance of the constituent terms.

$$\Gamma_D(x) = \sum_{i=1}^{n} \alpha_i(x)\, \Gamma_{G_i}(x) + \sum_{j=1}^{m} \beta_j(x)\, \Gamma_{e_j}(x)$$

α_i and β_j are membership functions such that

$$\sum_{i=1}^{n} \alpha_i(x) + \sum_{j=1}^{m} \beta_j(x) = 1$$

Fuzzy Reasoning, Fuzzy Linguistic Variable

In retreating from precision in the face of the overpowering complexity, it is natural to explore the use of what might be called linguistic variables; that is, variables whose values are not numbers but words or sentences in a natural or artifical language. The motivation for the use of words or sentences rather than numbers is that linguistic characterizations are, in general, less specific than numerical ones. For example, in speaking of age, when we say, "John is young," we are less precise than when we say "John is 25." Young may be regarded as a linguistic value of the variable Age, with the understanding that it plays the same role as the numerical value 25 but is less precise and hence less informative. The same is true of the linguistic values very young, not young, extremely young, not very young, etc. The values of the linguistic variable constitute its term-set. The term set of linguistic variable Age is T(Age) = {Young + not young + very young + not very young +...+ extremely young +... + old, not old, etc.} in which '+' is used to denote the union. The numerical variable Age,whose values are numerical,constitute the base variable for Age. Each linguistic value is interpreted as a label for a fuzzy restriction on the values of the base variable. The fuzzy restriction defines the linguistic value. In order to characterize a fuzzy restriction, Zadeh introduces a compatibility function which associates with each value of the base variable in the interval [0,1] representing its compatibility with the fuzzy restriction. Zadeh associates the linguistic variable with two rules: (a) A syntactic rule that specifies the manner in which the linguistic values which are in the term set of the variable may be generated. (b) A semantic rule which specifies a procedure for computing the meaning of any given linguistic value. The meaning of terms is both subjective and context-dependent.

> Definition: A variable is characterized by a triple (X,U,R(X;u)), in which X is the name of the variable, U is a universe of discourse (finite or infinite set), u is a generic name for the elements of U, and R(X;u) is a subset of U which represents a restriction on the values of u imposed by X. A variable is associated with an assignment equation x = U: R(X), which represents the assignment of a value u to x subject to the restriction R(X). Obviously, the assignment equation is satisfied if and only if $u \in R(X)$.

Zadeh introduces the concept of possibility distribution in the following way: Let F be a fuzzy subset of U characterized by a membership function Γ_F. F is a fuzzy restriction on X if F acts as an elastic constraint on the values that may be assigned to X. So X = u: $\Gamma_F(u)$, where $\Gamma_F(u)$ is interpreted as the degree to which the constraint represented by F is satisfied when u is assigned to X. In order to express that F plays the role of fuzzy restriction in relation to X, we write R(X) = F. We call this equation a relational assignment equation because it represents the assignment of a fuzzy set to the restriction associated with X.

Now, let us have a proposition p = X is F, where X is the name of an individual, and F is the name of a fuzzy subset of U. We express p, by R(A(X)) = F where A(X) is an implied attribute of X which takes values in U and signifies that we assign F to the fuzzy restrictions on the values of A(X).

> Definition: Let F be a fuzzy subset of a universe of discourse U which is characterized by its membership function Γ_F, with the grade of membership, $\Gamma_F(u)$, interpreted as the compatibility of u with the concept labeled F. Let X be a variable taking values in U, and let F act as a fuzzy restriction, R(X), associated with X. Then the proposition "X is F", translated into R(X) = F, associates a possibility distribution, Π_X, with X which is postulated to be equal to R(X); Π_X = R(X), so Π_X may be regarded as an interpretation of the concept of fuzzy restrictions.

The possibility distribution function associated with X is denoted by π_X and is defined as being numerically equal to the membership function of F, $\pi_X = \Gamma_F$. Thus $\pi_X(u)$, the possibility that X = u, is postulated to be equal to $\Gamma_F(u)$.

If p is a proposition p = X is F, which translates into the possibility $\Pi_{A(X)}$ = F where F and A(X) are as in previous definitions, then the information conveyed by p, I(p), may be identified with the possibility distribution $\Pi_{A(X)}$, of the fuzzy variable A(X). thus, $I(p) = \Pi_{A(X)}$ where $\Pi_{A(X)} = R(A(X)) = F$.

I will conclude here, by pointing out that Zadeh's possibility theory provides a basis for a more adequate meaning representation of the environment of actions, tasks, goals, etc. and for the manipulation of the fuzzy knowledge, which is the type of knowledge which underlies natural language as well as most of human reasoning.

If the goal of science and objective knowledge is to construct models that are closer and closer approximations of reality, Zadeh's fuzzy logic, that is a model for approximate reasoning with vague data is an enormous step forward: rather than regard human reasoning processes as themselves 'approximating,' to some more refined and exact logical process that could be carried out perfectly with mathematical precision, Zadeh has suggested that the essence and power of human reasoning is in its capacity to grasp and use inexact concepts directly. Zadeh argues that attempts to model or emulate it by formal systems of increasing precision will lead to decreasing validity and relevance. Most human reasoning is essentially "shallow" in nature and does not rely on long chains of inference unsupported by intermediate data. It requires, rather than merely allows, redundancy of data and path of reasoning; it accepts minor contradictions and contains their effects so that universal inferences may not be derived from their presence.

Part 10

FUZZY LOGICS: A SURVEY

H. M. Prade

FUZZY LOGICS : A SURVEY .

CONTENTS : p 85

FUZZY LOGICS : A SURVEY .

Henri M. Prade (*)

CONTENTS :

The topic of this talk (**) is "fuzzy logics" -- "logics" with an "s" because there are several manners to interpret the expression or for example , to define an implication . But enyway , it deals with some kinds of approximate reasoning .

Before beginning , I would want to say that this attempt of synthesis is the result of a common work with Didier Dubois , who is presently at Purdue University .

Tho first part of my talk is devoted to a rather philosophical introduction about principally fuzziness and causality . The second one deals with different kinds of multivalent logics , tho third one with what is called "generalized modus ponens" in fuzzy set theory , the fourth one with fuzzy-valued logics . Naturally , I will terminate by some concluding remarks .

* Visiting scholar at Stanford Art.Int.Lab. Stanford University ,94305 CA. Apr.1978
The author is supported by a scholarship of the Institut de Recherches en Informatique et Automatique . Rocquencourt . 78150 LE Chesnay . France .
** H.O.S. Cambridge . MA. 5/24/78 .

1. INTRODUCTION :

a) What is fuzziness ?

Fuzzy set theory was first introduced by L.A.Zadeh in 1965 [30] . Since this dato , there have been more than a thousand of papers about this domain and its applications . A lot of them dealed , in different manners , with set theory and logic .

Notwithstanding this big amount of works , there is some confusion still , about what is fuzziness .In fact , there are basically two points of view .

In the first one , introduced by Zadeh , we consider a universe of discourso U and a subset A where transition between membership and nonmembership is gradual than abrupt . There is no well-defined boundary for this fuzzy subset A . For each $u \in U$, $m_A(u) \in [0,1]$ expresses the compatibility of predicate A with the element u of U . m_A , mapping from U to [0,1] is the membership function of A . However , if imprecision -- especially subjective one -- is usually fuzzy , this fuzziness is different from imprecision in the sense of tolerance intervals . Linguistic description (it is to say , very often , summarized description of complex situations) are fuzzy in nature . More generally ,we can say perhaps that human perception is fuzzy . Let us consider an example : U = [50,220] , A = "tall" ; from the statement "X is tall" we may induce , as Zadeh pointed it out [31] , a distribution of possibility about the tallness of X : $m_A(u)$ can be interpreted as the degree of possibility that X's height is equal to u . We are not interested here in what way m_A is got , in fact , it can be shown that generally a precise knowledge of m_A is not important for the practical applications .

The second point of view , dual in some sense of the first one , was introduced by Sugeno in 1974 [27] . In Sugeno's approach we are interested in guessing (most often subjectively) if an a priori non-located element $o \in U$ is an element of a subset A of U ; A is not necessarily fuzzy here . $g_U(A)$ expresses the subjective degree of belief in the statement "$u \in A$" . g_U is a mapping from $\bar{P}(U)$ (set of the (fuzzy) subset of U) to [0,1] , g_U is called a fuzzy measure . The notion of fuzzy measure is very general , because only increasingness is supposed , not the additi_

vity . So , probability functions , the belief function and plausibility function studied by Shafer [75] , and even possibility fuoctions are particular cases of fuzzy measures .

Let me give an example to illustrate both points of view . When we are looking at a Ming vase , we can sny for example that it is big or beautiful ; here , "big" and "beautiful" are predicates which point to fuzzy subsets in tho sense of Zadeh of U , set of Ming vases . But we may also try to guess -- becauso we are not experts ,just amateurs -- if the vase is genuine or counterfeit . Both approaches are not exclusi_ ve : we may try to guess is the vase is old where "old" is obviously a fuzzy predi_ cate . Sometimes the difference between the two approaches is very subtle : when I say "X is tall" , either I may model the statement by "X is an element of the fuzzy subset of the tall men" or I may consider that it induces a possibility distribution which values the possibility that X's height is an element of some fuzzy or non-fuz_ zy interval -- for example [175,180] .

b) Implication & causality :

After having tried to explain what is fuzziness , I am going to end this introduc_ tion by some intuitivo considerations about fuzzy logic . It may be a calculus either on the levels of belief of precise propositions or on the truth of proposi_ tions involving fuzzy predicates . In both cases , multivalent logics can be the logical calculus underlying fuzziness . In most of the multivalent logics , there is no more excluded-middle law : this situation may be interpreted as either the absence of decisive belief in one of the sides of a precise alternative or tho interference between antonym fuzzy concepts (e.g. "small" and "tall") .

Fuzzy logics must also cope with difficult questions such as the difference between implication and causality . We must try to avoid confusion between deductive infe_ rence with fuzzy predicates and causal inference with dubious premisses or/and entailment (modelled by what is called "fuzzy relations") . In the five

following examples , the two first ones are implications and the other ones involve causality :

1) If the cat is black , he is not white .

2) If the cat is big , he is not small .

3) If the cat is run over by a car , he will be dead .

4) If the cat falls from the 10th floor , he will be more or less hurt .

5) If the cat overeats , perhaps he will become obese .

1 and 3 are not fuzzy , 2 ,4 and 5 are fuzzy in some sense .

2. MULTIVALENT LOGICS :

1)Description :

There are many possibilities to extend the definition of the usual connectives of the binary logic to multivalent logics . Most of them were first studied by Lukasiewicz in the thirties . I will first recall them .

a) Multivalent logics using "max" and "min" (resp.) for disjunction and conjunction (resp.) :

Bellman & Giertz [1] and Fung & Fu [10] have shown that to preserve structural properties such as associativity , commutativity , distributivity , idempotency for conjunction and disjunction connectives , and semantic properties such as continuity , growth for their interpretative functions , the only functions , when the valuation set is [0,1] , are :

$v(P \wedge q) = \min (v(P), v(Q))$

$v(P \vee q) = \max (v(P), v(Q))$

Most of the authors use for negation $v(\neg P) = 1 - v(P)$, but it is not the only choice which is compatible with involution property . Excluded-middle law is no more valid .

For the implication , many interpretative functions have been proposed :

1) $v(P \rightarrow Q) = \max (1 - v(P) , v(Q))$ (Dienes , Rescher [22])

2) $v(P \rightarrow Q) = \min (1 , 1 - v(P) + v(Q))$ (Zadeh's logic)

3) $v(P \rightarrow Q) = 1$ iff $v(P) \leq v(Q)$
$= v(Q)$ otherwise (Brouwer implication . See [11],[20],[24])

4) $v(P \rightarrow q) = \max (1 - v(P) , \min(v(P), v(Q)))$ (Zadeh)

5) $v(P \rightarrow Q) = \min (1 , v(Q)/v(P))$ (Goguen [13])

It will be too long to give here all the other classical connectives of these logics . For example , the equivalence connective which is associated with the implication number 2 is

$v(P \leftrightarrow Q) = | v(P) - v(Q) |$

b) Other multivalent logics :

There are other possibilities to define conjunction and disjunction :

* "probabilistic logic" :

$v(\neg P) = 1 - v(P)$

$v(P \wedge Q) = v(P).v(Q)$

$v(P \vee Q) = v(P) + v(Q) - v(P).v(Q)$

the corresponding implication is

$v(P \rightarrow Q) = 1 - v(P) + v(P).v(Q) = v(\neg P \vee Q)$

* non-distributive logic :

$v(\neg P) = 1 - v(P)$

$v(P \wedge Q) = \max (0 , v(P) + v(Q) - 1)$

$v(P \vee Q) = \min (1 , v(P) + v(Q))$

In this calculus , we have the excluded-middle law but no more the idempotency and distributivity properties . Here , $[0,1]$ is a non-distributive complemented lattice instead of a distributive non-complemented lattice with "miny and "max" operators . This logic is not very easy to interpret although it seems to be a candidate to model ambiguity (see [12] , [9]) .

More generally it is possible to build new connectives as , for example , $(1-\lambda).\max(v(P) ,v(Q)) + \lambda.\min(v(P) ,v(Q))$ where $\lambda \in [0,1]$, which has some very attractive properties and which , for $\lambda=0.5$, can model a connective "and/or" .

2) Hints for a comparison of these multivalent logics :

Some interesting points of view for a comparison are :

- Is Piaget's group (1) kept ? It is always the case : for example , with the implication number 4 , but it is kept with implications number 1 and 2 .

---

1 . Let 0 be a propositional variable containing elementary propositions P,Q,R,... joined with logical connectives . 0 is a wff symbolically written $0 = f(P,Q,R,...)$. Four transformations can be defined on 0 :
1) identity $I(0) = 0$; 2) negation $N(0) = \neg 0$; 3) converse $R(0) = f(\neg P, \neg Q, ...)$;
4) correlation $C(0) = \neg R(0)$.
These transformations ,for function compositional law , have a Klein group structure whose table is : (see bottom of the next page)

- associated set theory :

For example , the inclusion associated with implication 1 is :

$$A \subset B \iff \max(1 - m_A(u) , m_B(u)) \geq 0.5 \quad \forall u \in U .$$

the inclusion which corresponds to implication 2 is the classical definition introduced by Zadeh :

$$A \subset B \iff m_A(u) \leq m_B(u) \quad \forall u \in U .$$

- transitivity of implications :

For example , implication 1 satisfy :

$$v(P \rightarrow (Q \rightarrow R)) = v((P \wedge Q) \rightarrow R)$$

but 2 does not .

- modus ponens :

The modus ponens rule allows Q to be inferred from P and P→Q in propositional calculus . In multivalent logics the problem is to compute v(Q) given v(P) and v(P→Q) . Many authors [15] have looked for a detachment operation * such that :

$$v(P) * v(P \rightarrow Q) \leq v(Q)$$

to have v(Q) as large as possible . Some of them have proposed "min" for * , others the product . Wit' v(P→Q)= max(1 - v(P) ,v(Q)) , if min(v(P) ,v(P→Q)) ≥ 0.5 then min(v(P) , v(P→Q)) ≤ v(Q) ≤ max(v(P) , v(P→Q)) . Here , * = min and the validity of a chain of implications is equal to the validity of the least valid

	I	N	R	C
I	I	N	R	C
N	N	I	C	R
R	R	C	I	N
C	C	R	N	I

Piaget [21] established that , for children , learning of human reasoning demands a perception of these transformations that is to understand the difference between sentences such as :
"Good poets are bad husbands"
"Good poets are not bad husbands"
"Bad poets are good husbands"
"Bad poets are not good husbands"

implication in the chain . With $\star = .$, the validity decreases with the length of the chain (with implication 5 $v(P).v(P \rightarrow Q) \leq v(Q)$) .

In Dienes-Rescher logic , if P is called a tautology as soon as $v(P) \geq 0.5$, then every theorem of the standard propositional calculus is a tautology in this logic [16] ,[5] . Gaines [11] has shown that , in some sense , Dienes-Rescher logic was a fuzzification of the standard propositional calculus .

With $v(P \rightarrow Q) = \min(1 , 1 - v(P) + v(Q))$,

if $v(P) = \alpha$ and $v(P \rightarrow Q) = 1$, then $v(Q) \geq \alpha$

if $v(P) = \alpha$ and $v(P \rightarrow Q) = 1 - \epsilon < 1$, then $v(Q) = \alpha - \epsilon$.

At the end of n inferences whose truth values are equal to $1-\epsilon$, the truth value of the premise being α , the conclusion has a truth value equal to $\alpha - n\epsilon$.

3. GENERALIZED MODUS PONENS :

a) The problem :

In the precedent section , we were interested in manipulating statements as

(1) $P \equiv$ "$u \in A$" where A is a fuzzy subset on U and $v(P) = m_A(u)$ (for example "u is a tall man").

In this section , we consider statements as :

(2) $P \equiv$ "X is A" where A is a fuzzy subset on U which induces a possibility distribution $\pi_X = A$ (for example in "X i tall" , "tall" is a fuzzy subset on the universe of heights) . In fact , this statement can be viewed as equivalent to an infinity of statements $p \equiv$ "u is the height of X" with $v(p) = m_A(u)$, $\forall u \in U$.

For performing approximate reasoning with such statements as (2) , we need new modifier rules and new rules of inference .

b)Modifier rules :

(introduced by Zadeh [32] , [33])

If a modifier M (e.g. "not" , "very" , "more or less") is modelled by a function f (e.g. $1-(.)$, $(.)^2$, $(.)^{0.5}$) then M(P) has a possibility distribution $f(\pi)$ where π is the possibility distribution associated with the statement P .

Examples :

M("X is A") = "X is M[A] .

M("X is A and Y is B") = "(X,Y) is M[A.B]

where A.B is the joint of the fuzzy subsets A and B on U and V :

$$m_{A.B}(u,v) = \min(m_A(u), m_B(v))$$

c) Rules of inference :

The three following ones have been proposed by Bellman & Zadeh [2] ,[32] .

- the projection principle : Let $\pi_{(X1,\ldots,Xn)} = A$ a possibility distribution on

the cartesian product of universes U1.U2.....Un . π can be restricted on the subset U1.....Us of U1.....Un in two ways : either in fixing the "values" of Xs+1 ,..., Xn (conditional possibility distribution) or by projecting π on U1.....Us , taking the maximum value of π over Us+1.....Un (marginal possibility distribution) . Reciprocally $\pi_{(X1,...,Xm)}$ can be extended to U1.....Un by cylindri_ cal extension , i.e. $\pi_{(X1,...,Xn)}$ has the same values as $\pi_{(X1,...,Xm)}$ independently of the values of Xm+1 ,...,Xn .

- the particularization of $\pi_{(X1,...,Xn)} = A$ is the modification resulting from the stipulation that the possibility distribution $\pi_{(X1,...,Xm)}$ is B . The result is $\bar{A} \cap \bar{B}$ where $\bar{A}$ and $\bar{B}$ are the cylindrical extension of A and B on a common universe U1....Ul of which U1.....Un and U1.....Um are subsets .

- the entailment principle :

from $\pi_{(X1,...,Xn)} = A$, we can infer $\pi_{(X1,...,Xn)} = B \quad \forall B \supset A$.

What is called compositional rule of inference results of the application of both particularization and projection principles :

From $\pi_{(X,Y)} = A$ and $\pi_{(Y,Z)} = B$, we infer $\pi_{(X,Z)} = A \circ B$ where o denotes the max-min composition :

$$m_{A \circ B}(u,w) = \max_{v} \min(m_A(u,v), m_B(v,w))$$

a fuzzy subset on a cartesian product of universes is called a fuzzy relation between these universes . What is called generalized ponens is a particular case ot the compositional rule of inference where form $\pi_X = A'$ and $\pi_{(X,Y)} = R$, we infer $\pi_Y = A' \circ R$. Let us study now ,more particularly , this generalized modus ponens -- which is a kind of interpolation .

c) Generalized modus ponens :

$\pi_{(X,Y)} = R$ models here some fuzzy causality relation between fuzzy subsets of U and

V . Practically the problem is to build R from a rule such as "If X is A , then Y is B" where A and B are fuzzy subsets of U and V (e.g. A = "small" , B = "large" , U and V are some intervals of the real line appropriate to the nature of X and Y) . We have $m_R = f(m_A , m_B)$, where the question is what can we use for f ?

- we can use any of the implications introduced in section 2 .

e.g . $m_R(u,v) = \min(1 , 1 - m_A(u) + m_B(v))$

- mainly in practical applications , a lot of authors use more simply the joint of A and B : $m_k(u,v) = \min(m_A(u) , m_B(v))$.

If we denote by Ri the fuzzy relation built from implication number i , we heve the following result :

if $m_B(v) \in [0.5,1]$, then

$$m_{R3}(u,v) \geq m_{R2}(u,v) \geq m_{R5}(u,v) \geq m_{R1}(u,v) \geq m_{R4}(u,v) \geq \min(m_A(u),m_B(v))$$

If $B'i = A' \circ Ri$, the same inequalities hold for $m_{B'i}(v)$. The "largest" B'i , the "fuzziest" the result .

Some particular cases are interesting : [12]

A'= A , then $m_{B'i}(v) \geq m_B(v)$, $\forall v$; but B' coincides with B for the highest membership elements . Thus in approximate reasoning , generalized modus ponens provides more valid conclusions than standard modus ponens , for any Ri or the joint .

A rather funny particular case of modus ponens is the well-known rule of three . The classical rule is : If X is equal to a , and when X is equal to α , Y is equal to β , then Y is equal to $a\beta/\alpha$. But this rule can be extended with fuzzy numbers (i.e. convex and normalized fuzzy set of the real line) namely $\bar{a}$, $\bar{\alpha}$, $\bar{\beta}$ -- $\bar{x}$ models a number whose value is around x ; then $\bar{b} = \bar{a} \circledast (\bar{\beta} \% \bar{\alpha})$ where $\circledast$ and % denotes the extended multiplication and division (see [4] , [6]) ;

Here $m_R(u,v) = m_{\beta\%\alpha}(v/u)$ and we have $\bar{a} \circ R = \bar{a} \circledast (\bar{\beta} \% \bar{\alpha})$

N.B. : Very recently , Diaz [3] has proposed another way to build R from "If X is A , then Y is B" as :

$$m_R(u,v) = \begin{cases} \max(1 - m_A(u) , m_B(v)) & \text{if } m_A(u) \leq m_B(v) \\ \min(1 - m_A(u) , m_B(v)) & \text{if } m_A(u) > m_B(v) \end{cases}$$

This method has some very attractive properties .

More generally , practical cases involve several rules :

"X is A"

"If X is A1 , then Y is B1"

"....."

"If X is An , then Y is Bn"

then B = A o max (R1,...,Rn) .

Some interesting questions , which are not completely solved yet , are linked to this problem :

- consistency of the rules [3]
- non-redundancy between the rules
- given R , extract rules (see Tong [28]) .

4. FUZZY-VALUED LOGICS :

In all the considerations of the sections 2 and 3 , we have worked with ordinary fuzzy sets -- it is to say that the membership values were real numbers of [0,1] . Most of these works can be also performed in a similar , but generally more difficult way , with what is called fuzzy sets of type 2", i.e. fuzzy sets whose membership values are fuzzy subsets of [0,1] (generally fuzzy numbers) . I will just point out some features of these fuzzy-valued logics .

Warning : In the following paragraph , A , B are fuzzy sets of type 1 of [0,1] .

By means of Zadeh extension principle , the ordinary operation "min" and "max" on type 1 can be extended to the fuzzy subsets of type 2 [1] . Negation operation "1 - ." can be extended in the same way in "1 - ." ; $m_{1-A}(u) = m_A(1-u)$, 1 - A is the antonym of A . $\overline{max}$ and $\overline{min}$ denotes the extension of max and min . The practical rule of construction of $\overline{max}(A,B)$ and of $\overline{min}(A,B)$ is : take the left most part for the $\overline{min}$ and the right most part for the $\overline{max}$ of the pair of the increasing parts of A and B ,do the same for the decreasing parts . $\overline{max}$ and $\overline{min}$ are commutative associative , idempotent , distributive on each other , and verify De Morgan's law and absorption law .

So there is no problem to compute the truth value of the conjunction and of the disjunction of two propositions whose truth values are fuzzy sets of [0,1] , i.e. quasi linguistic because we can model truth values as "truth" , "false" , "very truth" , "borderline" as fuzzy sets of [0,1] .

Generalized modus ponens can be also extended by means of a $\overline{max}$ $\overline{min}$ composition :

$$\bar{m}_{AoR}(v) = \overline{\max}_{v}\ \overline{\min}_{A,R}\ (\bar{m}_A(u) , \bar{m}_R(u,v))$$

$\bar{m}_{AoR}(v)$, $\bar{m}_A(u)$, $\bar{m}_R(u,v)$ are here fuzzy subsets of [0,1] , and AoR , A ,R are fuzzy subsets of type 2 of V ,U , U V .

---

1. A binary operation * on real numbers is extended to fuzzy subsets A and B of the real line by the formula :

$$m_{A*B}(w) = \max_{w=u*v} \min(m_A(u), m_B(v))$$

CONCLUDING REMARKS :

I have not spoken of all the existing topics in fuzzy logics . I must mention still

- the works in switching logic by Kandel [14] and many others about the canonical form of a fuzzy expression involving max and min (extension of the minimization of a boolean function).
- fuzzy first order logic .
- the works by Shotch [26] or by Vaina [29] in fuzzy modal logic . Schotch has introduced an intermediary modal operator between the ordinary possibility and necessity operators which models : "it might be possible that" .
- in the precedent development , we consider for statements only truth values . Zadeh [32] ,[33] has proposed also to work with probabilistic values or even possibilistic values (which are modelled by subintervals of [0,1] , i.e. 0-fuzzy sets) . Sanchez [23] has introduced a modus ponens for possibility-valued statements .

Fuzzy logics has already been used in a lot of applications . I can quote , for examples :

- use of heuristic rules , modelled as a union of fuzzy relations in order to define a linguistic controler (see Mamdani [19])
- (medical) diagnosis (see Sanchez [24])
- in artificial intelligence , the language FUZZY , defined and implemented by LeFaivre [17] ,[18] to manipulate approximate knowledge .

I will conclude by saying that there is no universal or even canonical way for building fuzzy logics . Practical applications will winnow the chaff from the grain .

REFERENCES :

1.Bellman R.E., Giertz M.:"On the analytic formalism of the theory of fuzzy sats" Information and Sciences vol 5 pp 149-156 . 1973 .

2.Bellmam R.E. , Zadeh L.A. :"Local and fuzzy logics" International Symposium on Multivalued Logic .Indiana . May 1975 . 83 p .

3.Diaz Muchio :Pr.Zadeh's seminar . Berkeley . 5/9/78 .

4.Dubois Didier & Prade Henri :" Operations on Fuzzy Numbers " To be published in The Int.J.of Systems Science . 1978 .

5.Dubois Didier & Prade Henri "Fuzzy Logics and Fuzzy Control ." Submittad to The Int.J.of Man-Machine Studies .

6.Dubois Didier & Prade Henri :" Fuzzy Real Algebra : Some Results " Purdue University Memorandum TR-EE 78-13 Part A .1978 .

7.Dubois Didier & Prade Henri :" Operations in a Fuzzy-Valued Logic ." (21 p) Purdue University Memorandum TR-EE 78-13 ,Part D . Feb.1978 .

8.Dubois Didier & Prade Henri :"Comment on "Theory of Fuzzy Sets","Fuzzy Sets as a Basis for a Theory of Possibility","A Theory of Approximate Reasoning"&"PRUF-A Meaning Representation Language for Natural Languages" by L.A.ZADEH ." (20 p) Purdue University Memorandum TR-EE 78-13 ,Part E . Feb.1978 .

9.Dubois Didier & Prade Henri :" An Alternative Fuzzy Logic ." (12 p) Purdue University Memorandum TR-EE 78-13 ,Part F . Feb.1978 .

10.Fung L.W., Fu K.S.:"An axiomatic approach to rational decision making in a fuzzy environment" in Fuzzy Sets and their Applications to Cognitive and Decision Processes edited by Zadeh ,Fu ,Tanaka ,Shimura .Academic Press 1975 pp 227-255 .

11.Gaines B.R. :"Foundations of fuzzy reasoning" Inter. J. Man-Machine Stu. vol 8 pp 623-668 . 1976 .

12. Giles R.:"Lukasiewicz logic and fuzzy set theory " Inter. J. Man-Machine Stu. vol 8 pp 327-373 . 1976 .

13. Goguen J.A.:"The logic of inexact concepts" Synthese vol 19 pp 325-373 . 1968

14. Kandel A. :"On minimization of fuzzy functions" I.E.E.E. Trans. on Comp. vol C-22 n 9 pp 826-832 . 1973 .

15.Robert Kling :"Fuzzy-PLANNER : Reasoning with Inexact Concepts in a Procedural Problem-Solving Language ." J. of Cybernetics vol 4 n 2 . 1974 . pp 105-122 First version in vol 3 n 4 . 1973 .

16.Richard C.T.Lee : "Fuzzy Logic and the Resolution Principle ." J. of ACM vol 19 n 1 . pp 109-119 . 1972 .

17.Richard A.LeFaivre : "The Representation of Fuzzy Knowledge ." J. of Cybernetics vol 4 n 2 . pp 57-66 . 1974 .

18.Richard A.LeFaivre : "Fuzzy Problem Solving ." PhD Thesis Univ.of Wisconsin . 1974 . 187 p .

19.E.H.Mamdani : "Applications of Fuzzy Set Theory to Control System ." in "Fuzzy Automata and Decision Processes" Edited by M.M.Gupta , G.N.Saridis , B.R.Gaines . North Holland . 1977 . pp 77-88 .

20.Maydole R.E.:"Paradoxes and many valued set theory" J.of Philo. Logic vol 4 pp 269-291 .1975 .

21.[Piaget] Hermine Sinclair "Piaget's theory of development : the main stages" pp 68 -78 in "Critical features of Piaget's theory of tho development of thought" ed. by F.B.Murray . 1972 . Univ.of Delaware .

22. Rescher N.:"Many valued logic" New York Mac Grawhill .1969 .

23. Sanchez Elie : "On possibilirty qualifications in natural languages" memo. UCB/ERL M77/28 . 1977 .

24. Sanchez Elie : "Solutions in composite fuzzy relations equations :applications to medical diagnosis in brouwerian logic " pp221-234 in Fuzzy Automata and Decision Processes ed. by Gupta , Saridis , Gaines . North Holland . 1977 .

25. Shafer Glenn :"A mathematical theury of evidence" Princeton University Press 297 p . 1976.

26. Schotch Peter K.:"Fuzzy modal logic" Proc. of the Int. Symp. on Multiple-Valued Logic . I.E.E.E. 1975 . pp 176 -182 .

27. Sugeno M.: "Theory of fuzzy integrals and its applications" Thesis Tokyo Institute of Technology 124 p . 1974 .

28. Tong R.M.: "Analysis of fuzzy control algorithms using the relatiun matrix"

Int. J. Man-Machine Stu. vol 8 pp 679-696 . 1976

29. Vaina L. : "Semiotics of "with" ". To be published .

38. Zadeh Lotfi A. : "Fuzzy Sets ." Inf.& Cont. vol 8 pp 338-353 . 1965 .

31. Zadeh Lotfi A. : "Fuzzy sets as a basis for a theory of possibility. memo. UCB/ERL M77-12 . 1977 .

32. Zadeh Lotfi A. : "A theory of approximate reesoning" memo. UCB/ERL M77-58 1977.

33. Zadeh Lotfi A. : "PRUF- A Meaning Representation Language for Natural Languages UCB ERL Memo. M77/61 . 1977 .

Part II

Four Models for the Description of Systems

by

S. Cushing

FOUR MODELS FOR THE DESCRIPTION OF SYSTEMS

In this report we compare the expressive power and perspicuity of four notational frameworks for the specification and definition of systems and requirements. The control maps of Hamilton and Zeldin (1976c), the R-Nets of Alford et al. (1977), the commutative diagrams of mathematical category theory (Arbib and Manes, 1975), and a modified version of the R-Net framework are reviewed and the relationships among them discussed. The relative merits of the four frameworks along the two dimensions of expressive capacity and clarity or convenience of use are evaluated, and recommendations are made for the conditions under which each framework might profitably be used.

1. REVIEW OF CATEGORY-THEORETIC RESULTS ABOUT HOS

Cushing (1978a) analyzes the three primitive control structures of Higher Order Software (HOS) in terms of the arrow language of mathematical category theory and proves the following theorems:

Theorem 1: HOS composition is identical to category-theoretic composition, as far as the mappings involved are concerned.

Theorem 2: HOS set-partition and class-partition are category-theoretic duals, in that the commutative diagrams they imply differ only in the directions of corresponding arrows.

The proof of Theorem 1 is trivial, and the proof of Theorem 2 involves showing that the commutative diagrams implied by set-partition and class-partition are identical, with one minor difference in each case, to those used in category theory to define the coproduct and product, respectively, of two sets. The single minor difference involved is a reflection of the fact that the category-theoretic diagrams are used to define ways of combining sets, while the HOS-implied diagrams are used to define ways of combining functions; it has no impact at all on the proof of Theorem 2.

These considerations motivate the following three definitions:

Definition 1: Let f, g, and h be functions such that f is the parent in an HOS composition in which g and h, in that order, are the offspring. Then f is said to be the (functional) composition of g and h.

Definition 2: Let f, g, and h be functions such that f is the parent in an HOS set-partition in which g and h are the offspring. Then f is said to be the (functional) coproduct of g and h.

Definition 3: Let f, g, and h be functions such that f is the parent in an HOS class-partition in which g and h are the offspring. Then f is said to be the (functional) product of g and h.

Note that the order of g and h is crucial in Definition 1, but does not matter in Definitions 2 or 3. This fact follows from the symmetry of each of the coproduct and product diagrams and the asymmetry of the composition diagram. The

definitions, like the primitive control structures themselves, can be generalized in a straightforward manner to include cases in which more than two offspring are involved.

2. FUNCTIONS AND CONTROL STRUCTURES IN REQUIREMENTS NETWORKS (R-NETS)

The Software Requirements Engineering Methodology (SREM) is an "integrated, structured approach to requirements engineering activities" (p. 1-1) developed by Alford et al. (1977). On the face of it, SREM seems to share many features with HOS: it "begins with the translation and decomposition of system level requirements; performs analysis, definition, and validation of the software requirements; and ends with documentation of the software requirements..." (p. 1-1). It includes "a set of software support tools...to automate many of the previously manual activities associated with requirements engineering," including a "structured, formal Requirements Specification Language (RSL)."

The role played by control maps in HOS--i.e., the description of data[1] and control relationships--is played in SREM by what are called requirements networks or R-Nets. An R-Net represents

> the order of logical processing steps that must be performed. An R-Net may contain Ands, Ors, and For Each Nodes; it must be enabled and terminated. The processing steps are alphas or subnets which may be expanded into lower levels of detail. An R-Net may also contain validation-points, events, and interfaces (p. D-14).

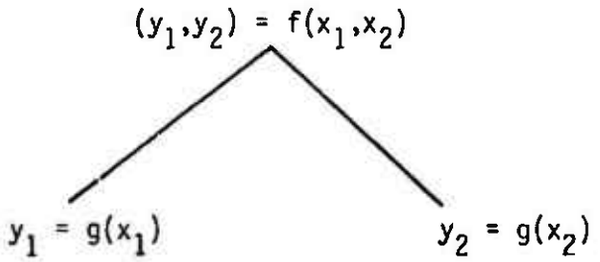
An alpha, in this definition, is "a processing step in the functional requirements domain" (p. D-2), and a subnet represents "the order of logical processing steps that must be performed in order to perform the requirements of the processing step that represents it at the next higher-level" (p. D-16). A validation-point is "a logical point in the processing at which timing, value, or presence data must be obtainable in the real time software in order to validate that the requirements have been fulfilled" (p. D-21), and an event is "an identified point that exists in the processing of one or more R-Nets or subnets and which may cause the enablement of an R-Net" (p. D-7). Interfaces are of two kinds: an input-interface is "a 'port' between the data processing system and the rest of the BMD system which accepts data from the other system (e.g., the

[1] Strictly speaking, data flow is not explicitly indicated on an R-Net, but it seems to be deducible from the information that is explicitly represented, as we will see.

radar-returns)" (p. D-9); an output-interface is "a 'port' between the data processing system and the rest of the BMD system which transmits data to the other system (e.g., the radar-commands)" (p. D-12).

A sample R-Net, containing each of these basic elements and some others, is exhibited in Figure 1 (p. D-34). To determine the relationship between this R-Net and the HOS control map for the same system, we have to ask how the system represented in the R-Net could be expressed explicitly in terms of data types, functions, and control structures. Since the data types in this example are not explicitly specified in the figure[2], we will ignore them for the time being and concentrate for now entirely on the functions and control structures that do, it seems, appear in the R-Net.

The first thing we notice is that some of the elements of the R-Net become superfluous, when viewed in terms of HOS. The R-Net Start, Terminate, and Input- and Output-Interface elements are all unnecessary, because the notions they represent are implicit in the conventional notation for a function. In the control map



for example, we know automatically that the function starts being evaluated as soon as the inputs x_1 and x_2 are available and terminates as soon as the outputs y_1 and y_2 are available. Nothing further need be said about starting or terminating the processing of the system. We also need say nothing about whether inputs come from or outputs go to other parts of the data processing system or other parts of the BMD system, because this will be clear from the input-output relationships represented in the complete BMD control map. Inputs come from wherever they are produced as outputs, and outputs go to wherever

---

[2]See Footnote 1.

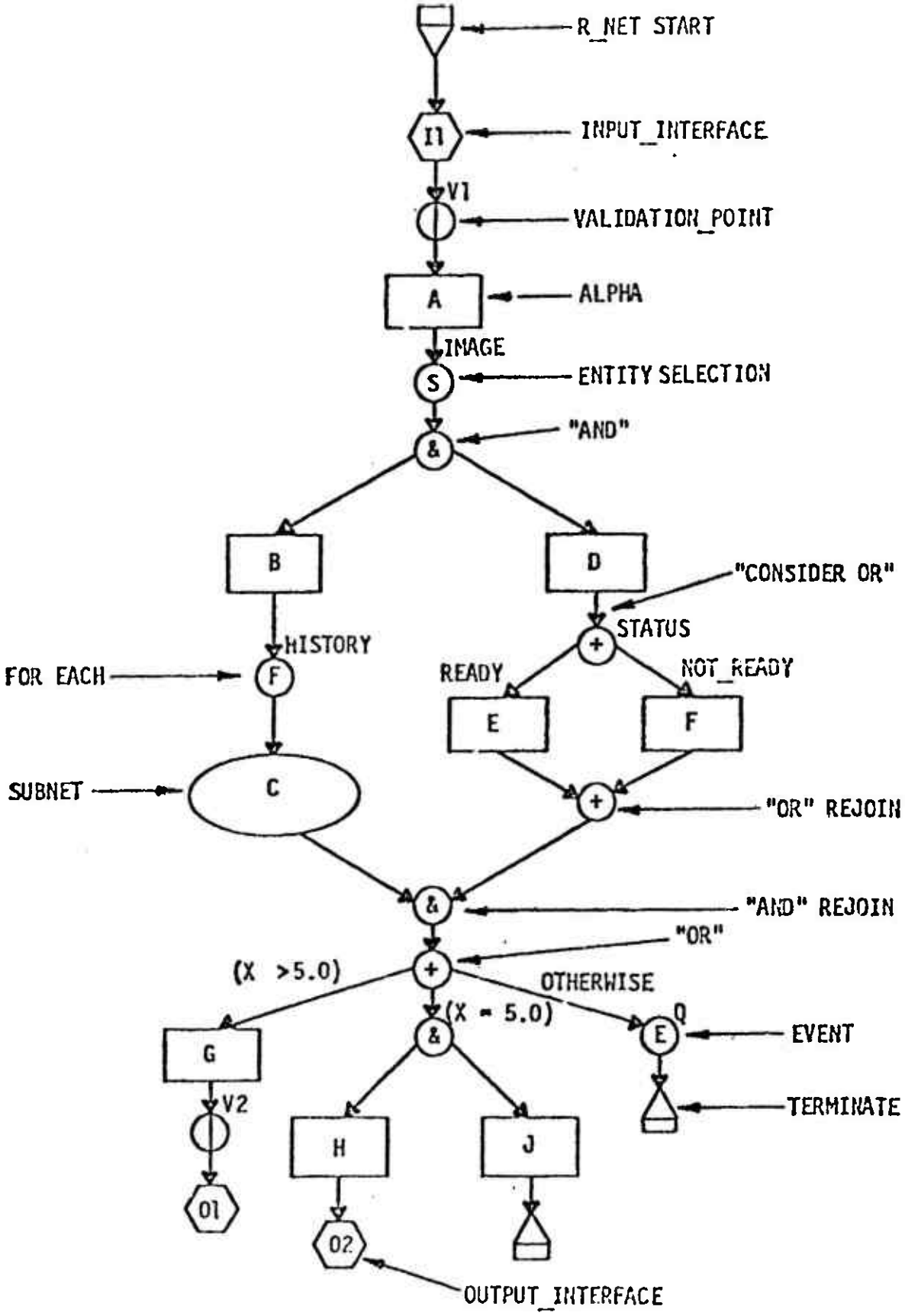


Figure 1: Sample R-Net Structure in Graphical Form

they are used as inputs. Output and input are relative notions and, again, nothing further need be said on the matter (given the complete control map).

The functions represented in the R-Net clearly include the alphas--A, B, D, E, F, G, H, and J--, but they also include the Validation Point V1, the Entity Selection S, the Subnet C, and the Event Q. Each of these performs a function, in the mathematical sense, and thus would be represented as a function in HOS. The Validation Point, for example, really just denotes a test, which can be represented in HOS as a set partition or coproduct (Definitinn 2); the Subnet, similarly, can be written as a control map in itself, since, whatever its internal structure may be, its overall effect is to evaluate a function.

The control structures represented in the R-Net include sequential flow, denoted by "→", and, denoted by "&", or, denoted by "+", and for each, denoted by "For Each" in the figure. Consider or in the figure can be subsumed under or, for our present purposes, and, as we will see in Section 4, we can also ignore the for each node. Finally, we will incorporate V1, A, and S into a single function, denoted by "A", since they are all joined sequentially by "→" and thus form a single composed function in the mathematical sense.

This gives us the skeletal R-Net structure exhibited in Figure 2, in which all of the simplifications we have just discussed here have been made and only the functions and control structures that appear explicitly in the R-Net are shown. The first thing we notice about this skeletal R-Net structure, which still contains, it must be emphasized, all of the essential mathematical components of the original R-Net (except the For Each node), is that the HOS nntinn of level of decomposition (Hamilton and Zeldin, 1976a) is nowhere to be seen. The sort of level that we saw mentioned above in the definition of subnet corresponds roughly to the HOS notion of layer, which we will not discuss here (but see Hamilton and Zeldin, 1976b,1977a,b). One purpose of HOS decomposition levels, as opposed to layers, is to make explicit the full set of structural relationships that hold among all of the functions that get performed by a system. Higher-level functions get performed precisely by requiring their lower-level functions to get performed instead, a form of delegating responsibility, so to

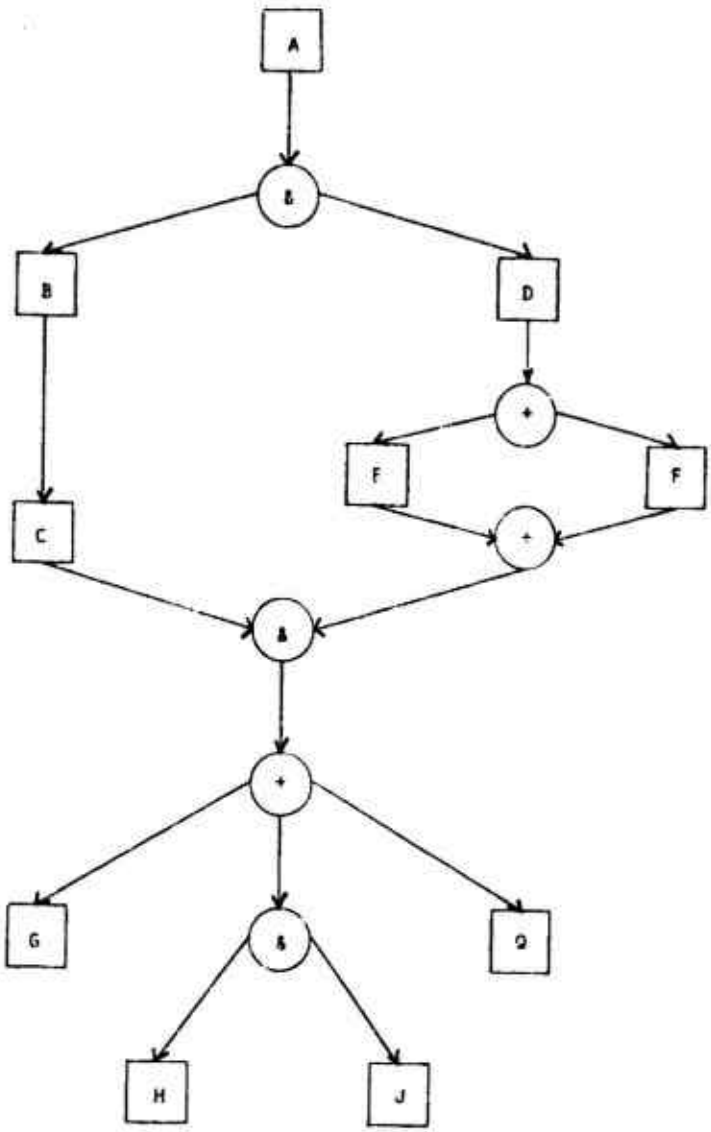
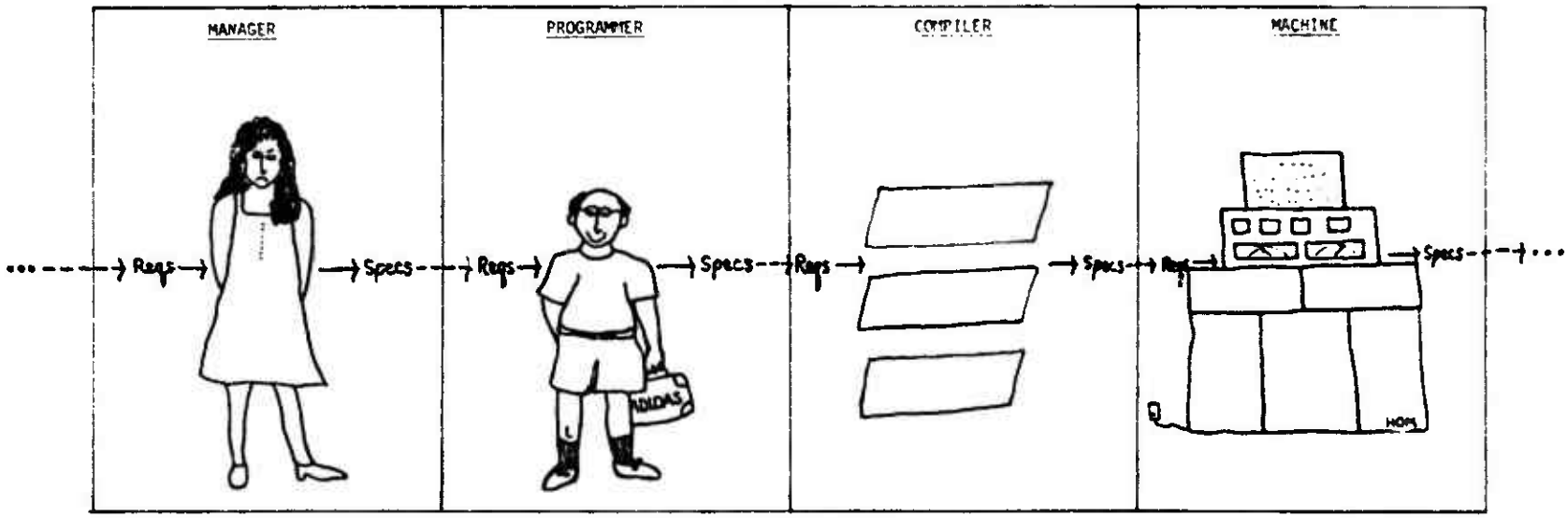


Figure 2: Skeletal Structure of Sample R-Net Showing Only Explicit Functions and Control Structures

speak, to use a convenient anthropomorphic metaphor (Cushing, 1977, 1978b). Lower-level functions, in turn, get themselves performed precisely in order to fulfill the requirements set by their higher-level functions. This relative notion of requirements bears some stressing, because it seems that the idea of requirements becomes hopelessly confusing and unintelligible without it. To use a simplified example (see Figure 3), the manager's specifications--i.e., his instructions--become the programmer's requirements; the programmer's specifications--i.e., his program--become the compiler's requirements; the compiler's specifications--i.e., the machine code it produces--become the computer's requirements; and the computer's specifications become part of someone else's requirements, depending on the circumstances. It is for this reason that HOS uses the same language and formalism for representing both specifications and requirements; the distinction between the two notions is not an absolute, but a relative one, depending entirely on one's point of view in a given context.

In Figure 2, for example, each of A and B performs a function, but so does their joint action. This higher-level function is their composition, in the sense of Definition 1, but is nowhere to be seen (explicitly) in Figure 2. Each of E and F performs a function, but so does their joint action, i.e., their coproduct, in the sense of Definition 2. The latter, higher-level function, however, is again nowhere indicated, as an actual function existing in its own right, in the R-Net structure of Figure 2. These two higher-level functions, furthermore, also combine, according to the diagram, to form a still higher-level function: the product of the composition of A and B and the coproduct of E and F. Again, one would never suspect that this function was present, as an existing function requiring to be performed, just from the information in the R-Net, because higher-level functions are not indicated explicitly in such diagrams.

We can incorporate these higher levels into an R-Net structure by determining what higher-level functions are involved and drawing boxes around them, just as there are boxes around the explicitly appearing functions, in this case, A, B, C, D, E, F, G, H, J, and Q. In the case of Figure 2, we have already determined that these higher-level functions include the composition of B and C, the coproduct of E and F, and the product of this composition and this coproduct.



Solid Arrow (⟶) denotes performance of function (processing step): actual change from input data to output data

Dashed Arrow (- - - -⟶) denotes transition between points of view: change only in status from specifications to requirements

Figure 3: Each Component of a System Produces Specifications from Requirements: One Component's Specifications Are the Next Component's Requirements

It is also easy to see that this R-Net contains, as well, the product of H and J; the coproduct of G, this product, and Q; and the composition of A and each of the highest-level functions we have determined so far. If we draw boxes, as suggested, around each of these higher-level functions, and assign each new box a name, denoting the higher-level function the box represents, we get the diagram exhibited in Figure 4. Note that the system as a whole contains five levels of box embedding, which correspond to five levels of decomposition in an HOS control map.

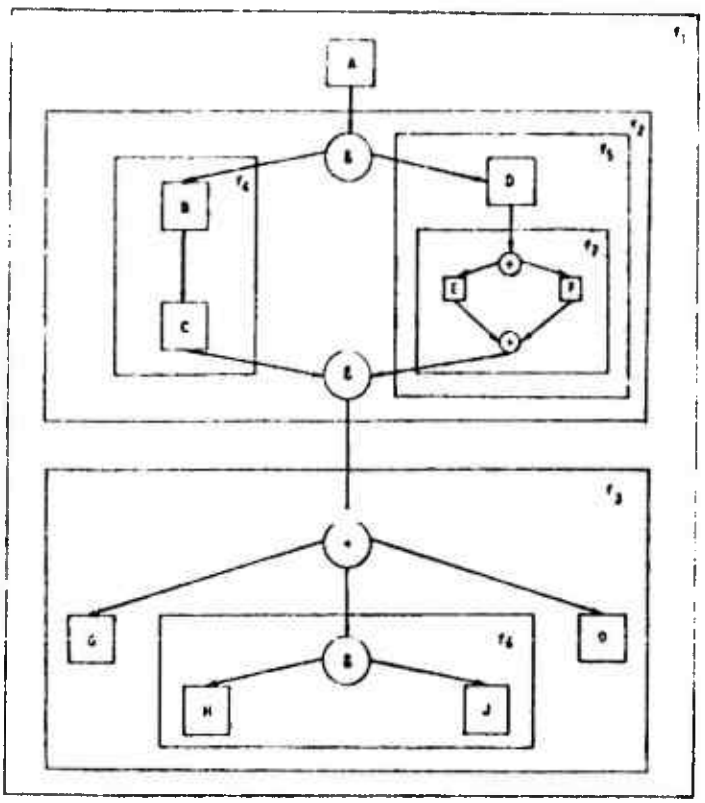


Figure 4: Skeletal Structure of Sample R-Net Showing All Higher-Level Functions as well as Explicit Ones

3. R-NETS AND CONTROL MAPS

We are now in a position to translate the R-Net structure into a control map. The process is, at this point, fairly simple and mechanical, but going through the steps graphically should help to put the relationship between the two systems notations into sharper relief. First we have to make certain that each + and & control structure receives a complete graphic description, in the sense that each such node is correlated with a matching Rejoin node. The only nodes in Figure 4 that are incomplete, in this sense, are those corresponding to f_3 and f_6, and completing them results in the diagram of Figure 5. The dotted circles are included only to identify the added Rejoin nodes and have no further significance. Note also that a new sequential flow arrow has had to be added to connect these new nodes.

Now that we have guaranteed that every + and & node is graphically complete, by adding the necessary Rejoin nodes, we reverse ourselves and eliminate the redundancy involved in having two nodes for each control structure. Physically, this means moving each + or & node and its matching Rejoin node toward each other until they meet and then merging them into a single node. This process results, in this case, in the diagram exhibited in Figure 6. Note that each & and + node is connected now to the relevant boxes by lines, rather than arrows, since no flow is involved in & or + themselves. It is part of the meaning of "&", in other words, that both the f_4 function and the f_5 function, for example, get performed, and it is part of the meaning of "+", similarly, that either E or F, for example, get performed. Flow occurs only where the arrows actually appear now: from B to C in f_4, from D to f_7 in f_5, and so on. Note also that the graphical complication resulting from the fact that the + node in f_3 connects three, rather than two, boxes is entirely one of how best to draw the graph, not one of the functions or control structures themselves.

The diagram in Figure 6, like the original simplified R-Net structure in Figure 2, says that first we perform function A, then both function B followed by function C and function D followed by either function E or function F, and then either function G, both function H and function J, or function Q. It also contains, however, the higher-level structures that relate these functions, and

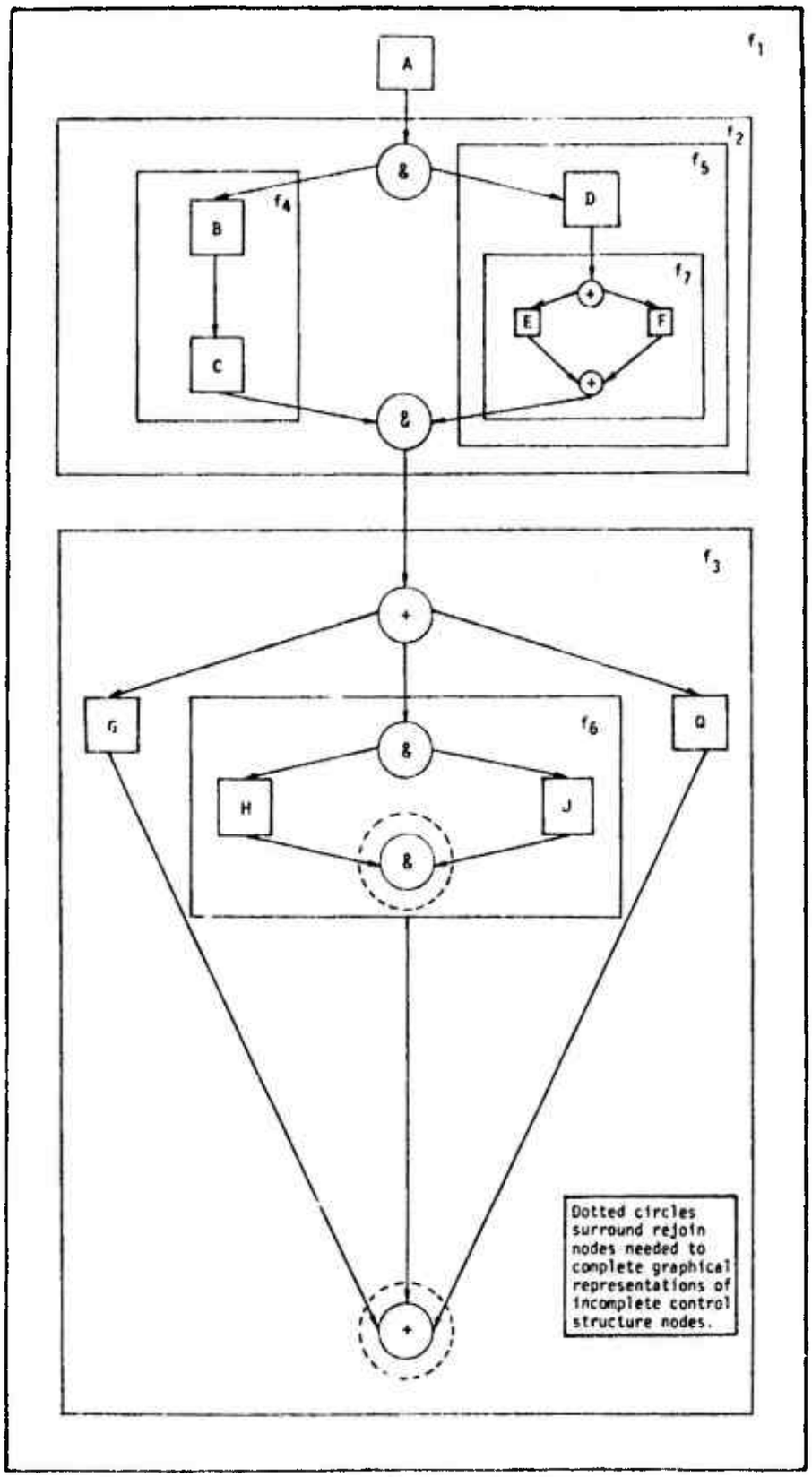


Figure 5: Skeletal Structure of Sample R-Net with All Control Structures Completed

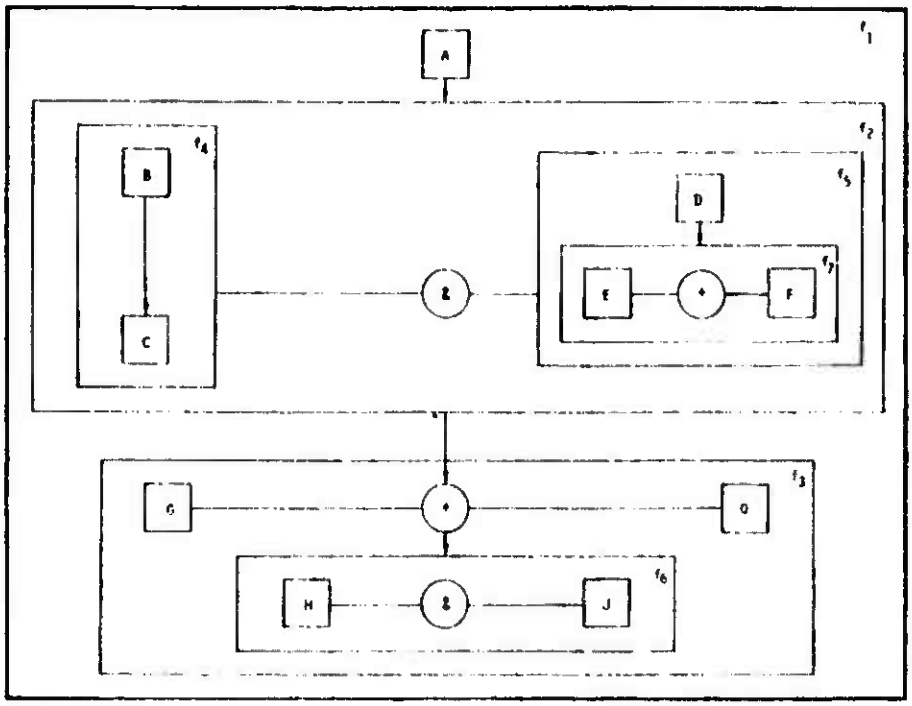


Figure 6: Skeletal R-Net Structure with Redundant Node, Eliminated

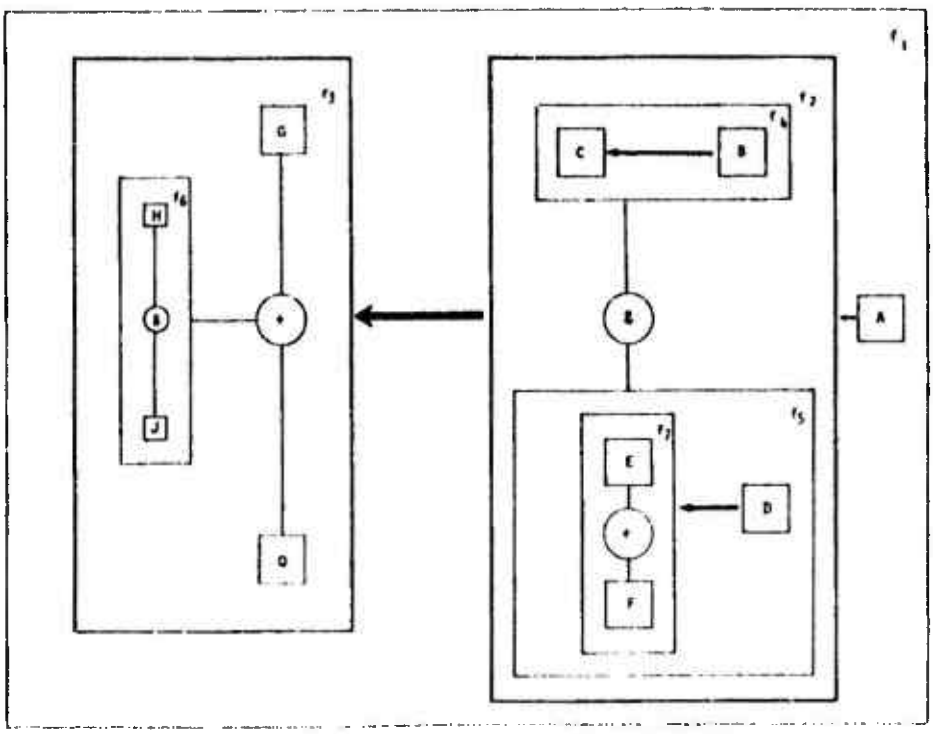
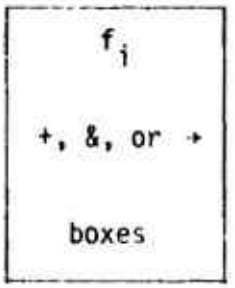


Figure 7 Skeletal R-Net Structure Reoriented by 90° Clockwise Rotation

it embodies all of this information without the redundancy of duplicate nodes for & and + control structures.

The next step is to turn the diagram on its side. If we take the diagram in Figure 6 and rotate it clockwise ninety degrees, we end up with the diagram in Figure 7 (with the labels suitably relocated). The reason for this reorientation is that it places the various functions involved more in line with where they will appear in the final control map.

Having now reoriented the diagram as a whole, the next thing we do is to rearrange each box internally, so that each function and control structure is located at the top center of its box. Each higher-level box should then have the internal structure



as indicated in Figure 8.

By now it should be perfectly clear what is going on, because the control map is virtually staring us in the face. Only two more steps, in fact, are necessary to get the overall functional structure of the control map from the diagram in Figure 8. First we connect each function name by a straight line to the boxes that occur with it in its own box, and then we erase the borders of the boxes themselves. The first step results in the diagram in Figure 9, and the second step results in the diagram in Figure 10.

As we have observed in connection with Figure 6, Figure 10 also contains all of the information about functions and control that is contained in Figure 2, plus the additional information about higher-level functions that HOS shows it is necessary to include. The overall function that is performed by the system, according to Figure 10, is f_1, which consists of A followed by f_2 followed by f_3.

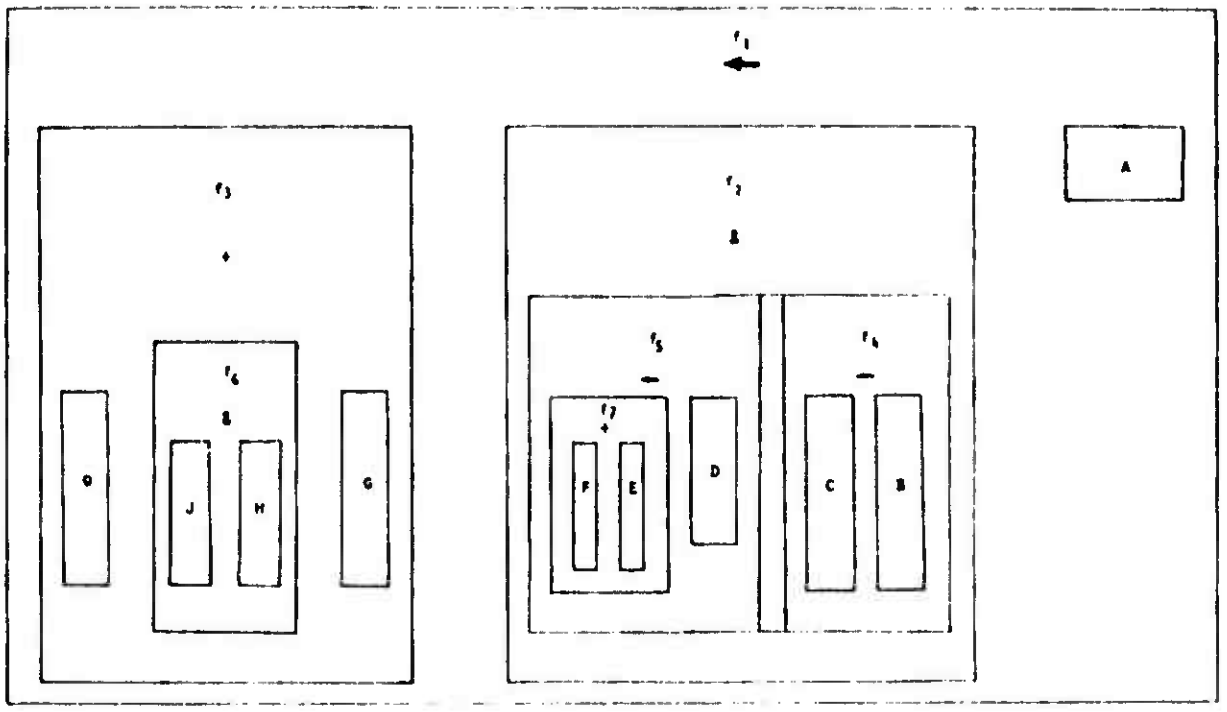


Figure 8 Skeletal R-Net Structure with Rearrangement of Internal Box Structure

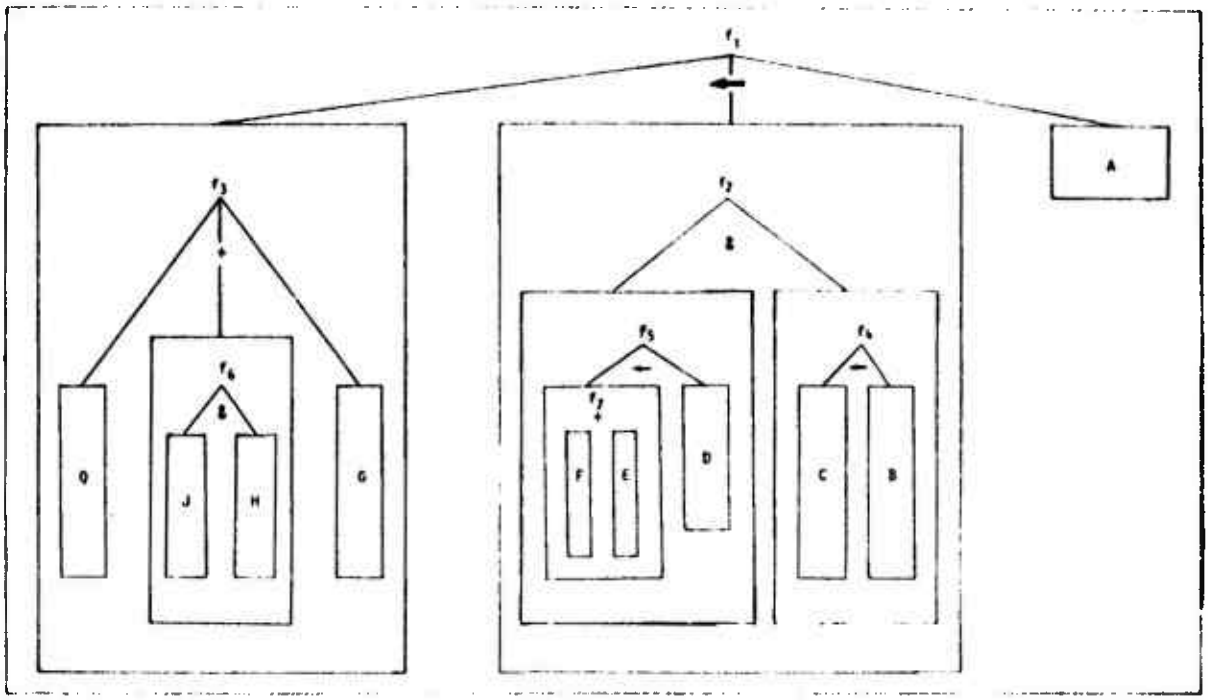


Figure 9 Rearranged Skeletal R-Net Structure with HDS Control Lines Added

Function f_2 is performed by performing both f_4, which consists of function B followed by function C, and f_5, which consists of function D followed by function 7, i.e., either function E or function F. Performing f_3, in turn, means performing either function G, function f_6, which consists of both function H and function J, or function Q. All of this information can be squeezed out of the R-Net structure, exactly as we have done here, but it is not included explicitly in the R-Net structure itself.

One reason we want this information to be included in the graphical representations of systems is, among other things, that it makes possible the formulation of principles that guarantee the correctness of interfaces, namely, the six HOS axioms (Hamilton and Zeldin 1974, 1976c). Higher-level functions are not represented in R-Nets because they do not entail any processing beyond that involved in the lowest-level functions that make them up (Alford, personal communication). "A Requirements Net, or R-Net", after all, "is used to describe the required flow of processing in response to a single stimulus which enables the net" (p.3-19). The problem with this, however, is that we have to recognize higher-level functional structure in order to get the relationships among processing steps straight.

This is a familiar situation in science. Such forces as gravity and such entities as electrons were first posited in physics not because they were directly observed, but because the behavior of matter that was observed could not be explained readily without them. Only by stepping back from the hard data and constructing abstract theories was it possible to make sense out of the data themselves. Here, similarly, the bottom line that we are interested in is the processing steps that actually get performed by the system. One of the main things we want to know about these processing steps, however, is whether their interfaces are correct, since, if they are not, the system will ultimately fail. Just as the positing of gravitation and electrons in physics enables us to explain important aspects of matter behavior, the explicit recognition of higher levels of functional structure enables us to solve this crucial problem of data processing. Once a system is specified in control map form, the six HOS axioms tell us which interfaces are correct, which are not correct, and how to fix the latter. The interfaces among the terminal nodes of a control map tree are correct, if (and only if) the interfaces among all the nodes in the tree are correct. Even if we are

interested, in other words, only in the functions that actually constitute processing steps, we still have to work out their higher-level functional structure in order to check that these lowest-level functions are actually interacting the way they should.

A second reason for requiring an adequate notational framework to provide a way of representing higher-level functional structure is that this makes possible the formulation of a notion of abstract control structure, which, in turn, enables us to go well beyond the complexity of systems that can be given simple descriptions entirely in terms of &, +, and →. An abstract control structure, in essence, is simply the relationship among functions that is represented by a control map in which one or more of the nodes is left variable (see Hamilton and Zeldin, 1976c). Once such a structure has been defined, we can use it to simplify control maps considerably by using its name in place of the portion of the control map that defines it. Depending on the complexity of the abstract control structures we bother to define, some extremely complex systems can be given very simple descriptions. What makes all of this possible, however, is precisely the ability to represent higher-level functional structure that control maps provide us with.

We can make the tree structure in Figure 10 look even more like the customary HOS control map by replacing "&", "+" and "→" with their approximate[3] HOS equivalents, respectively, "INCLUDE", "OR", and "JOIN". If we adopt the convention that "JOIN" denotes right-to-left flow, this gives us the diagram in Figure 11. As Cushing (1978a) points out, furthermore, control flow in a control map is given automatically by the specification of data flow, as long as only primitive control structures are involved, so the names of the control structures are superfluous in that case, from a theoretical point of view, once data flow has been indicated. Control structure names are useful in practice, however, because the name of a control structure serves as a check on whether the specified data flow is allowable or not, given a library of pre-defined control structures. This becomes especially important when abstract control structures

[3] "&", "+", and "→" seem actually to correspond to the non-primitive HOS control structures COINCLUDE, COOR, and COJOIN, but this difference need not distract us here. Lack of explicit data flow specification in R-Nets makes strict comparison with control maps only approximate in any case. See note 1.

are introduced and the library of control structures grows beyond the three primitives.

Let us now introduce data flow into the skeletal control map structure of Figure 11 to see exactly how this relationship between data and control flow emerges. The first function that gets performed in the system, other than the overall system function f_1 itself, is A, so we know that it is function A, on this level, that gets the system input data itself as its input. We will denote this input data by the symbol "input-list" to indicate that more than one variable is likely to be involved. We also know that it is the output of A that gets used as input to f_2 and the output of f_2 that gets input to f_3, because of the very meaning of "JOIN" (or of "→"), and our convention on right-to-left flow. Function f_3, furthermore, is the function on this level that produces the system output data as its output, since it is f_3 that completes the decomposition of f_1, once A and f_2 are given. We will call this data "output-list", since, again, more than one variable is most likely to be involved. This gives us the partial control map exhibited in Figure 12 as a description of data and control flow on the top two levels of our control map. Note the name "local-data", which we will use, with primes, to indicate data that gets used only within the system described by this control map.

Next we observe that f_2 gets performed by performing both f_4 and f_5, so the list of variables that is input to f_2 must be partitioned between the input of f_4 and the input of f_5, and the output of f_2 must be divided between the output of f_4 and the output of f_5. Since f_2 inputs local-data, we call the inputs to f_4 and f_5, respectively, "local-data$_2$" and "local-data$_1$", indicating that the variables in these two lists collectively form the full set of variables in local data.[4] Similarly, since f_2 outputs local-data', we will call the outputs of f_4 and f_5, respectively, "local-data$_2$'" and "local-data$_1$'", for essentially the same reason. Since f_4 is the composition of B and C, we know that the input of B must be the same as that of f_4 - namely, local-data$_2$ -

[4]The present author prefers to subscript functions in a control map from right to left, but data from left to right. There is no theoretical significance to this convention, and no confusion should arise, as long as it is understood and kept in mind. Actually, the convention is useful psychologically to undermine the erroneous notion that function and data subscripts must be related in some way, but, again, it is not necessary.

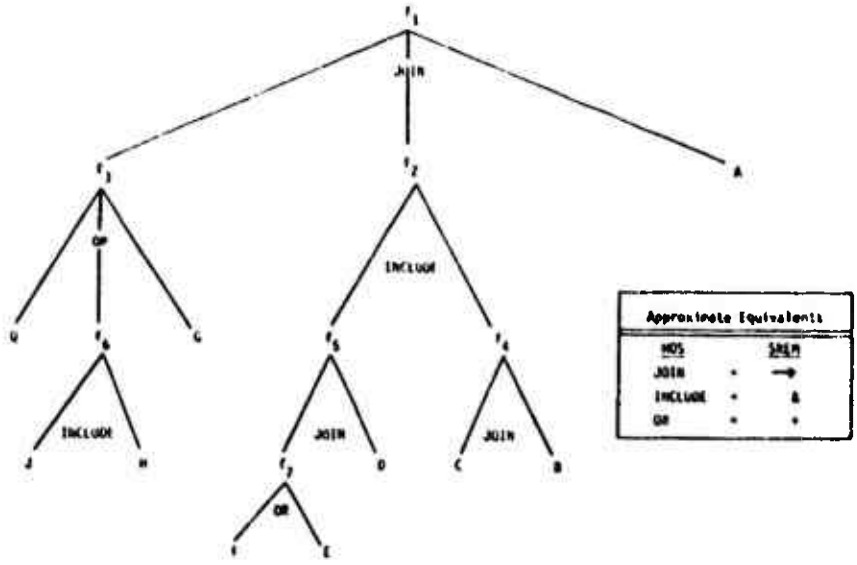


Figure 11 Skeletal Control Map Structure with HOS Names for Control Structures

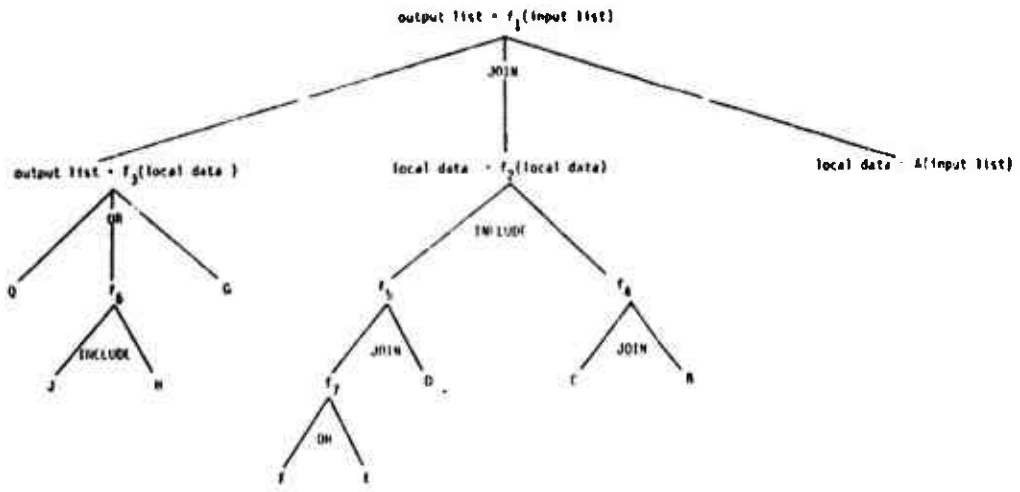


Figure 12 Data and Control Flow on Top Two Levels of Control Map

and the output of C must be the same as that of f_4 - namely, local-data$_2$'; we must also introduce a new local variable, say, "local-data'''" for the communication of B and C, i.e., output of B and input of C. Function f_5, similarly, is the composition of D and f_7, so similar considerations apply. Function D inputs local-data$_1$, function f_7 outputs local-data$_1$', and a new variable, say, "local-data''''", must be introduced for the output of D and the input of f_7. Finally (for this subtree), f_7 is the coproduct of E and F, so we have to partition the input and output sets of f_7, rather than the input and output lists, as was the case with f_2. Since f_7 inputs local-data''' and outputs local-data$_1$', we denote the inputs of E and F, respectively, by "^{2}local-data''''" and "^{1}local-data''''" to indicate that the union of the two input sets must be the set of local-data''', and we denote the outputs of E and F, respectively, by "^{2}local-data$_1$'" and "^{1}local-data$_1$'" to indicate that the union of the two output sets must be the set of local-data$_1$'. This gives us the data and control flow indicated in the partial control map exhibited in Figure 13. Note that we have reintroduced the condition (Figure 1) that determines the set partition involved in the decomposition of f_7, expressing it as a boolean-valued function of the relevant input data.

The only part of the tree remaining to be completed now is the two lower-left levels, as indicated in Figure 13. Since f_3 is the coproduct of Q, f_6, and G, and since it inputs local-data' and outputs output-list, we know that the input sets of Q, f_6, and G must form an exhaustive and mutually exclusive partition of the input set of f_3, and that their output sets must form an exhaustive, possibly nnn-mutually exclusive (Cushing, 1978) partition of its output set. In accordance with these facts, we choose to denote the inputs of Q, f_6, and G, respectively, by the symbols "^{1}local-data'", "^{2}local-data'", and "^{3}local-data'", and their outputs, respectively, by "^{1}output list", "^{2}output list", and "^{3}output list". Since f_6 is the product of J and H, we know that the input and output lists of f_6 must be partitioned to get the input and output lists of J and H. Accordingly, we denote the inputs of J and H, respectively, by "^{2}local-data$_1$'" and "^{2}local-data$_2$'", and their outputs, respectively, by "^{2}output-list$_1$" and "^{2}output-list$_2$" (See Hamilton and Zeldin, 1976a and Cushing, 1978a for a review of the superscripting and subscripting conventions we have just finished using.) . The resulting completely specified control map is exhibited in Figure 14. Note that, again, we have reintroduced the

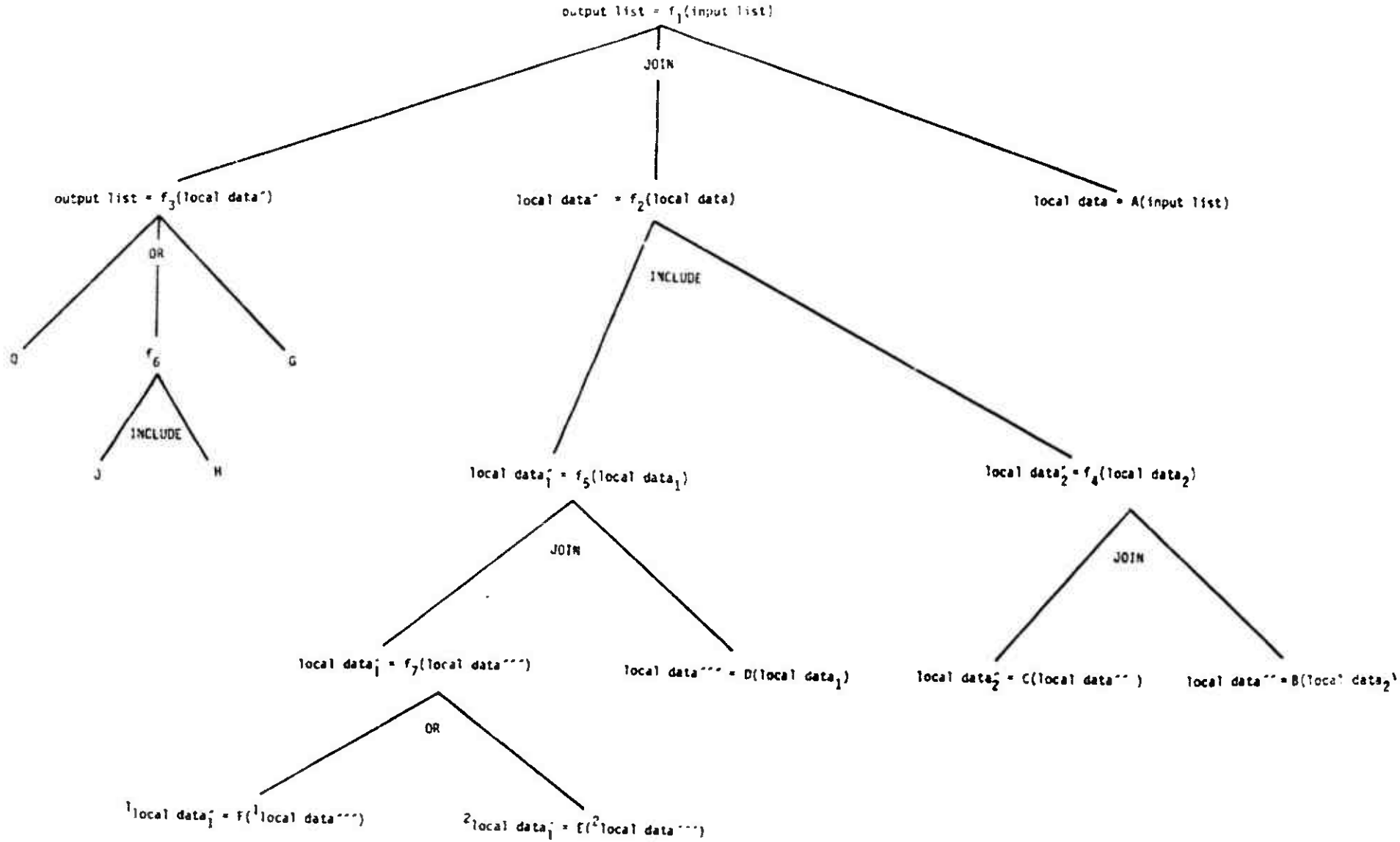


Figure 13: Data and Control Flow on All But Two Lower-Left Levels of Control Map

condition indicated in Figure 1 as determining the set partition, expressing it, again, as a boolean-valued function of the relevant input data.

Now that the data flow has been completely specified in the control map, we observe that the control structure names are superfluous. Communication between subfunctions automatically means a JOIN (i.e., composition) control structure, partitioning of the input and output sets automatically means an OR (i.e., coproduct) control structure, and partitioning the input and output lists automatically means an INCLUDE (i.e., product) control structure. If we remove the control structure names from the tree structure in Figure 14, then we get the control map exhibited in Figure 15, which, it follows, contains exactly the same information.

Each of Figures 14 and 15 contains all of the information contained in Figure 2, as we have shown, and also a lot more. Figures 14 and 15 contain explicit information on data flow, decomposition levels, and modularization which is present only implicitly in Figure 2. In particular, only the primitive functions A, B, C, D, E, F, G, H, J, and Q are explicitly represented in the R-Net structure, whereas all of the higher-level functions, crucial to a complete modular (correct interfaces) account of the system's functional structure, are included, along with the primitive functions, in the control maps.

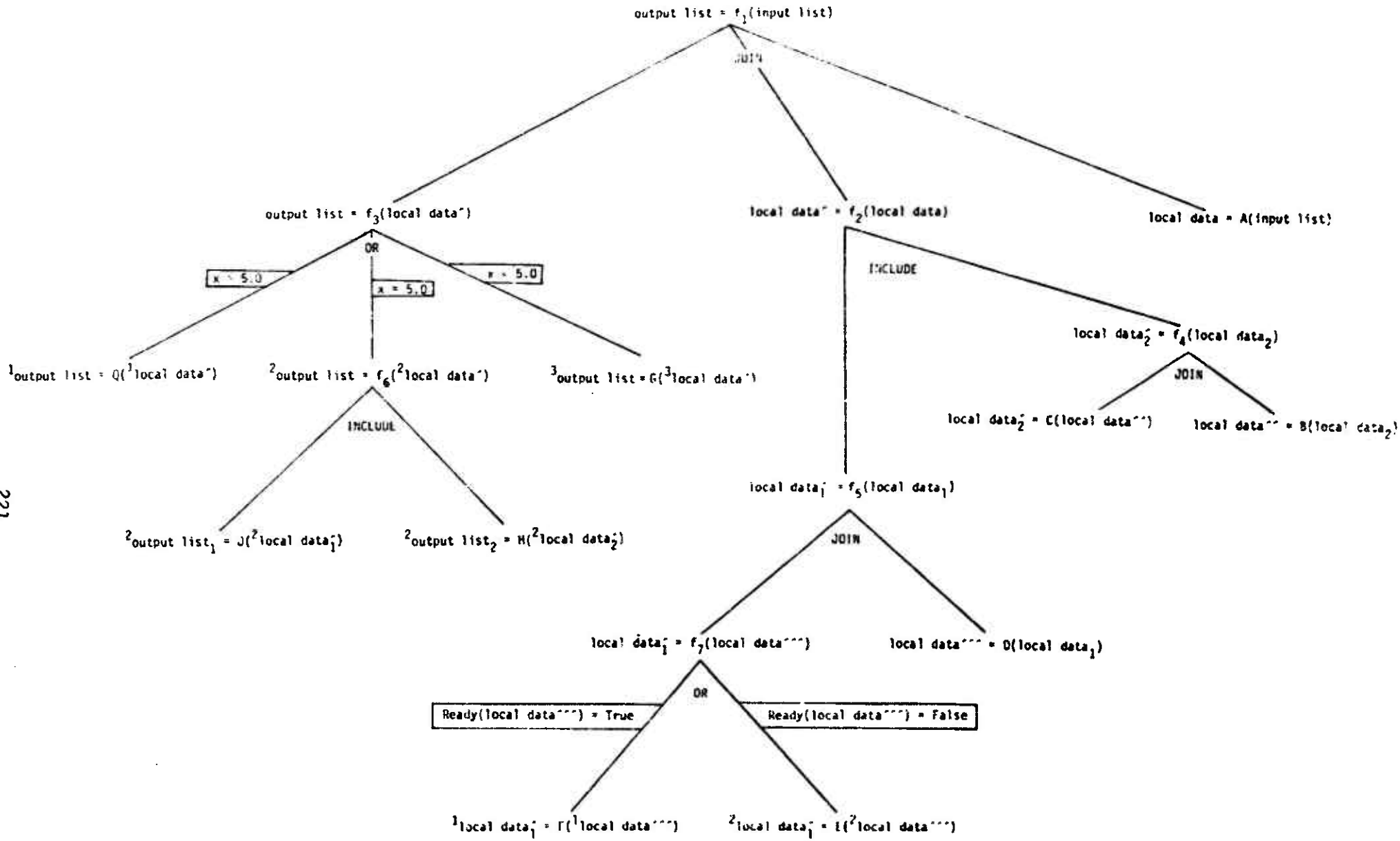


Figure 14: Complete Control Map with All Data and Control Flow Explicitly Indicated

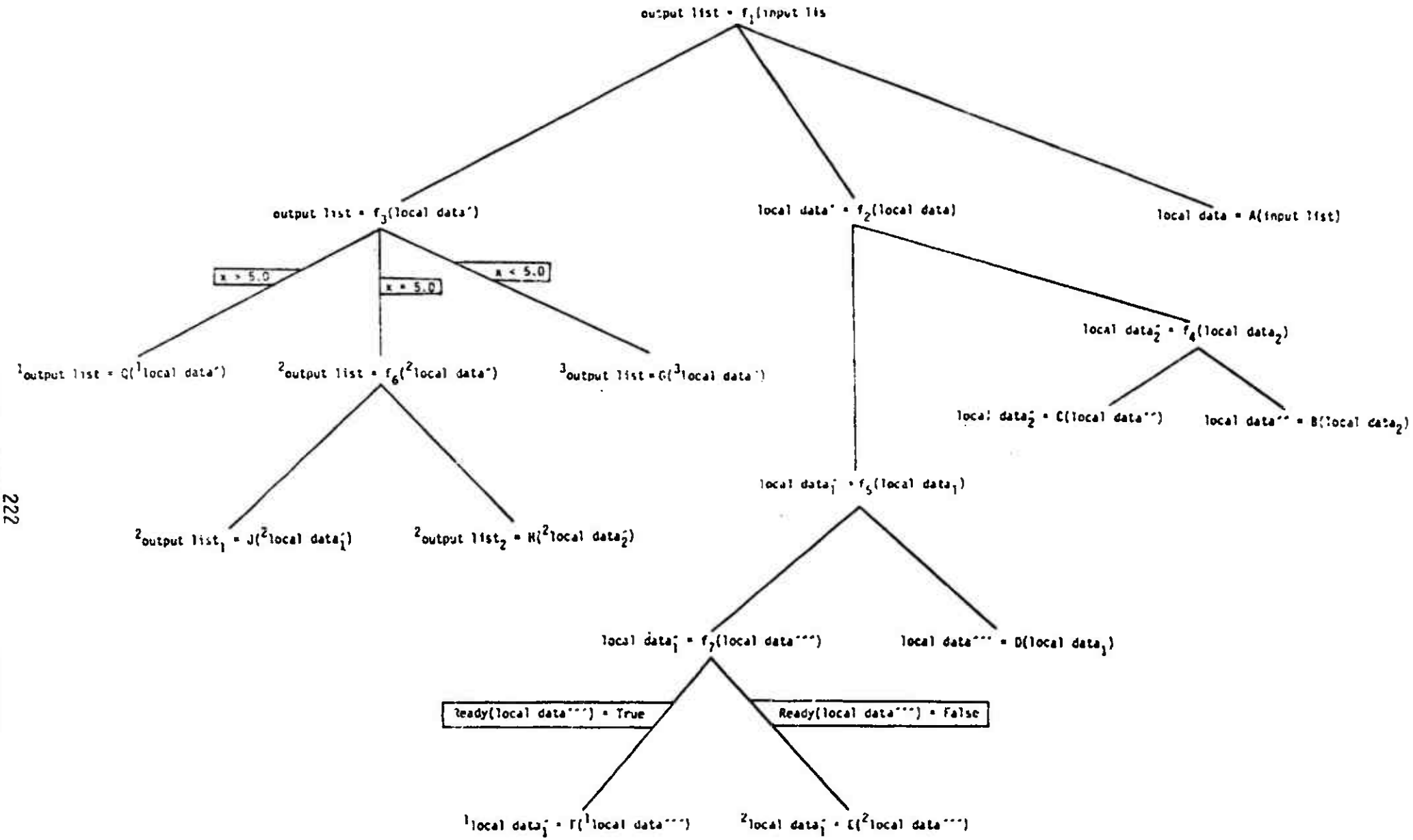


Figure 15: Complete Control Map with Control Flow Given by the Specification of Data Flow

4. TRANSLATION BETWEEN R-NETS AND CONTROL MAPS

In Section 3 we investigated the relationship between R-Nets and control maps by translating an idealized skeletal R-Net structure into a functionally equivalent[5] control map. Since the skeletal R-Net structure we examined contains all of the essential features (other than For Each nodes) of an actual R-Net, at least as long as an R-Net is viewed as a specification of control relations among functions, we can legitimately conclude that we have proven the following theorem:

> Theorem 3: For any R-Net that does not contain For Each nodes, there is a functionally equivalent control map that contains only primitive control structures.

Since our proof was strictly mechanical, requiring no special insight once the steps are learned, and since the R-Net structure we used has no special properties, but is fully representative of such R-Net structures in general, we can also conclude that we have proven the following theorem, as well:

> Theorem 4: The translation procedure from R-Nets to control maps is effective.

In other words, the translation process from R-Net structures to control maps is completely general, applying to all such structures in the same way without exception.

The converse of each of these theorems, furthermore, is also easily seen to be true, since the mechanical translation process from R-Nets to control maps is readily reversed. This gives us two additional theorems, as follows:

> Theorem 5: For any control map that contains only primitive control structures, there is a functionally equivalent R-Net that contains no For Each nodes.

---

[5] By saying that two functional control specifications are functionally equivalent, we mean that they perform the same function and contain the same functions and that the latter stand in the same control relationships in both cases. See Cushing 1977, 1978b, where the same notion is used in the proof of a different theorem.

Theorem 6: The translation procedure from control maps to R-Nets is effective.

As we have observed, however, a complete control map contains a lot of additional information regarding data flow and partitioning that is not explicitly represented in the corresponding R-Net. This information is lost in the translation process from control maps to R-Nets.

Information explicitly represented in the control map concerning higher-level functional structure, while not actually lost in the translation from control maps to R-Nets, is encoded in the R-Nets in a very non-perspicuous form. The boxes we drew in Section 2 to indicate this additional structure, in other words, are not generally included in an actual R-Net and, in any event, do not seem to be a particularly enlightening or convenient way to represent this higher-level structure (though they were obviously very useful in proving our four theorems). Whether there is a natural, revealing, and convenient notation for representing this structure within the R-Net framework remains to be seen, but it is far from evident, on the face of it, what such a notation would be.

Combining our four new theorems with some older results about HOS enables us to draw one further conclusion from this discussion. Hamilton and Zeldin (1974, 1976c) show that in order to have correct interfaces, i.e., in order even to qualify as a real system in the first place, a candidate system must satisfy the six HOS axioms. They also show, furthermore, that in order to satisfy the six axioms a system must be representable by a control map which contains only primitive control structures, i.e., JOIN, OR, and INCLUDE. In conjunction with Theorems 5 and 6, however, these facts give us the following result:

Corollary: Any system that can be represented by an R-Net with For Each nodes can be represented by an R-Net without For Each nodes and there is an effective procedure for eliminating the For Each nodes in going from the former to the latter.

Hamilton and Zeldin's results show that any genuine system at all can be represented by a control map that contains only primitive control structures, so we

know that any system that can he represented by an R-Net with For Each nodes can be represented by a control map that contains only primitive control structures. Theorems 5 and 6, however, tell us that a control map that contains only primitive control structures can be effectively rewritten as an R-Net without For Each nodes, so we know that the original R-Net with For Each nodes can be effectively rewritten that way as well by first translating into a control map with only primitive control structures.

One gap in this proof of the corollary, of course, is our assumption that For Each nodes are a legitimate construct in the first place. We have not shown, that is, that For Each nodes are even capable of appearing in a system at all without fouling up the interface relationships in the system. The best way to prove this assumption, and thus to fill the gap in the proof of the corollary, would be to specify the notion of For Each node explicitly as an HOS abstract control structure, but there is a very good reason for not trying to construct the relevant control map here. If the assumption is false and such a control map cannot be constructed, then not only is the corollary false, but the whole issue simply dissolves, since R-Nets with For Each nodes would then not describe any real systems. One has no business using For Each nodes in the first place, in other words, if one cannot show that they interact properly with the other three R-Net control structures. We are willing here to accept the legitimacy of For Each nodes for the sake of argument and in order for the corollary to have some suhstance, but the burden of proof rests with those who want to use R-Nets with For Each nodes as a tool for specifying systems. What the corollary tells us is that, if For Each nodes are legitimate at all, then they are superfluous, except perhaps as a convenient abbreviatory device. If they are not legitimate, then we cannot use them, and if they are legitimate, then we need not use them, so there seems little point in trying to resolve the question here.

If, for some reason, one does decide to use R-Nets to specify systems, then our theorems and corollary provide a useful means of checking the R-Nets we construct. Suppose we have an R-Net and we want to know whether the interfaces among the various modules represented and, in particular, among the processing steps are all correct. All we have to do, then, to check this is to translate the R-Net into a control map, as illustrated in Sections 2 and 3,

and then see whether all the control structures are either primitive or definiable in terms of the primitives. This tells us that the HOS axioms are satisfied by the control map and thus that the interfaces in the control map, and therefore also in the R-Net itself, are, in fact, correct.

5. ON EVALUATING NOTATIONAL FRAMEWORKS

Having investigated in some detail the relationship between R-Nets and control maps, we turn now to an examination of the relationship between each of these notational frameworks and that of commutative diagrams. As is the case in any domain, the value of a notational framework in system specification and requirements definition depends on the use we want to put it to. As long as our systems are relatively small and our requirements are simple, a notational framework like that of R-Nets may be just what we need, since description in terms of primitive control structures on one level of decomposition may suffice for our purposes in those cases. As soon as our systems become very large, however, the restriction leads to extremely cumbersome, unnecessarily complex specifications and, indeed, may even obstruct the development process.

Control map notation provides a way out of this situation by supporting in a natural way both the representation of levels of decomposition and the definition of abstract control structures. Control maps have, in fact, been found in practice to be very useful tools both in designing new systems (e.g., Harel, 1977) and in gaining insight into how systems already designed are supposed to work (e.g., HOS,1977). As we saw in Cushing (1978a), this notational framework is not perfect either, however, since it was only by representing the three primitive control structures by commutative diagrams that we were able to discover the category-theoretic duality of two of them, as expressed in Theorem 2.

The point, again, is that the value of a notational framework depends on its intended use. A theoretician interested in deep mathematical generalizations will use whatever notational system will enable him to discover those generalizations, sometimes control maps, sometimes commutative diagrams, and sometimes something else. An engineer looking for a practical tool for designing large real-world systems that work, however, would be well-advised to stick fairly exclusively with control maps, a notational framework that emerged directly out of an analysis of the properties of large real-world systems. We have already begun to appreciate the validity of this advice, as illustrated by the advantages we have found of control maps over R-Nets, and we will appreciate it even more when we examine the relationship between control maps and commutative diagrams.

6. Control Maps and Commutative Diagrams

Figure 16 shows the correspondences between the HOS primitive control structures written as control maps and the commutative diagrams that they imply, as these correspondences were used in the proofs (Cushing,1978a) of Theorems 1 and 2. Descriptions of the three control structures written in the standard notation of first-order predicate logic are also included to clarify the meanings of the diagrams even further for those who are familiar with that notation, and a clarification of the notational correspondences involved is also given.

The first question we have to ask now is how we might go about introducing a way of representing higher-level functional structure in commutative diagrams like those in Figure 16. In control maps, let us recall, higher-level functional structure becomes representable as a result of the fact that repeated decomposability is a natural consequence of the notation. Given any function, such as the one in Figure 17(a), we can decompose it into subfunctions, using one of the primitive control structures, to get a control map like the one in Figure 17(b). Each of these (sub)functions can then be decomposed, if we like, to get a control map like the one in Figure 17(c). Clearly, this decomposition process can go on for as long as we choose, resulting in control maps of systems with any number of levels of functional structure.

In all of the control maps in Figure 17, f is the overall system function, but which other functions are primitive functions, like the A, B, C, D, E, F, G, H, J, and Q of Figure 4, and which are higher-level functions, like the f_1 through f_7 of that figure, depends on how many levels of decomposition are involved. In (a) of Figure 17, if we view the figure as a control map at all, f itself is treated as a primitive function, whereas in (b) it is g and h that are the primitive operations of the system. In (c), (d), and (e) g and h are higher-level functions, along with f and k in (d) and both k and n in (e). The primitive operations in these three control maps are i, j, k and l in (c), i, j, m, n, and l in (d), and i, j, m, o, p, and l in (e).

Note also that higher-level functional structure can be represented abstractly in a control map without necessarily <u>having</u> to specify which primitive control

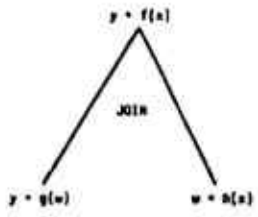
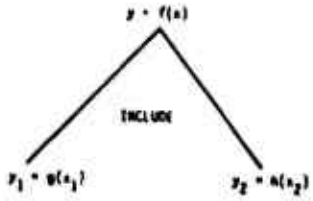
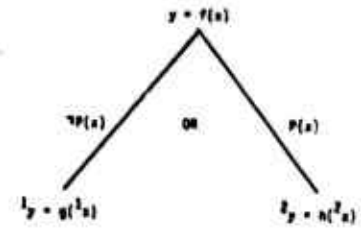
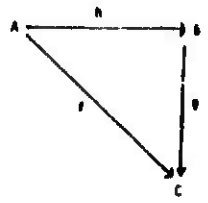
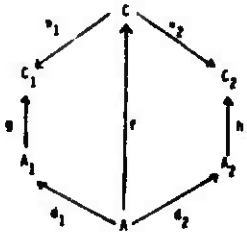
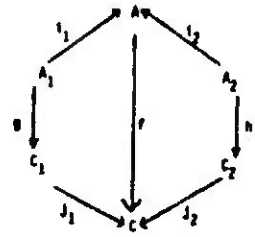
	(a)	(b)	(c)
(i) Control Maps:			
(ii) Commutative Diagrams:			
(iii) First-Order Logic (with Equality):	$(\forall x \in A)\ (f(x) = g(h(x)))$	$(\forall x \in A)\ (x = (x_1, x_2) \supset f(x) = (f(x_1), f(x_2)))$	$(\forall x \in A)\ ((x \in A_1 \supset f(x) = g(x)) \wedge (x \in A_2 \supset f(x) = h(x)))$
(iv) Notational Clarification:	x, w, y are variables of any type A = domain (f) = domain (h) B = range (h) = domain (g) C = range (g) = range (f) $x \in A$ $w \in B$ $y \in C$	x is a list of variables x_1, x_2 of any type y is a list of variables y_1, y_2 of any type A = domain (f) = domain (d_1) = domain (d_2) A_1 = range (d_1) = domain (g) A_2 = range (d_2) = domain (h) C_1 = range (g) = range (r_1) C_2 = range (h) = range (r_2) C = domain (r_1) = domain (r_2) = range (f) $x \in A$ $x_1 \in A_1$ $x_2 \in A_2$ $y_1 \in C_1$ $y_2 \in C_2$ $y \in C$ $A = A_1 \times A_2$ $C = C_1 \times C_2$	x, y are variables of any type ^1x, ^2x are variables of the same type as x ^1y, ^2y are variables of the same type as y A = domain (f) = range (i_1) = range (i_2) A_1 = domain (i_1) = domain (g) A_2 = domain (i_2) = domain (h) C_1 = range (g) = domain (J_1) C_2 = range (h) = domain (J_2) C = range (J_1) = range (J_2) = range (f) $x \in A$ $^1x \in A_1$ $^2x \in A_2$ $^1y \in C_1$ $^2y \in C_2$ $y \in C$ $A = A_1 \cup A_2$ $C = C_1 \cup C_2$ $A_1 \cap A_2 = \emptyset$

Figure 16: The HOS Primitive Control Structures Expressed as Control Maps, as Commutative Diagrams, and in First-Order Logic

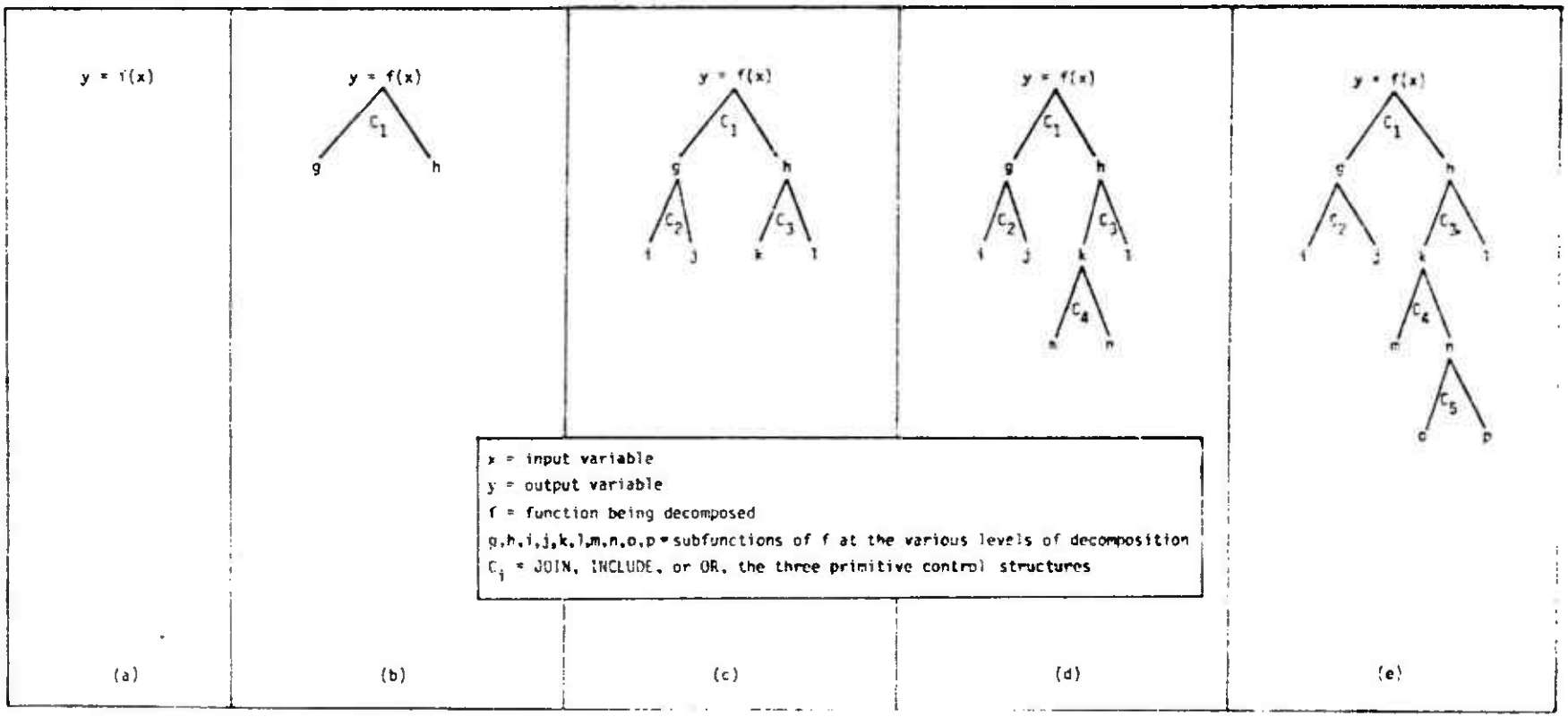


Figure 17: Representation of Higher-Level Functional Structure is Made Possible in Control Maps by Repeated Decomposability and Is Uniform for All Three Primitive Control Structures

structure is involved. Naturally a complete system description must include a full statement of all of the control structures involved, so the value of each "C_i" in Figure 17 would have to be filled in to obtain such a description, along with the relevant variables. Our point here, however, is that the representation of functional decomposition is uniform for all three primitive control structures, since they all get represented by downward extension of subfunctions (or "branching", in the tree-geometric, but not the computer-programming sense of that term).

The situation with commutative diagrams is very different in this respect, as we have already begun to see. We have seen, in Figure 17, that, while the arrow diagrams for INCLUDE and OR are reasonably similar in form (in fact, category-theoretic duals), the arrow diagram for JOIN looks rather different. Whereas each of (iib) and (iic) of that figure requires subsidiary arrows - i.e., π_k, d_k or i_k, j_k - to make all of the mapping relationships explicit, the diagram (iia) requires only the principal arrows f, g, and h denoting the overall function and its subfunctions. This discrepancy obviously has implications for the possibility of uniform decomposability, as the next two figures clearly illustrate.

For simplicity, let us assume that all instances of "C_i" in Figure 17 denote the same primitive control structure. Figure 18 shows what Figure 17 translates into in arrow-language terms when we take C_i = JOIN, for all i of the C_i in Figure 17, and Figure 19 shows what happens when we take C_i = INCLUDE, for all i of the C_i in Figure 17. The commutative diagrams that result when we take C_i = OR, for all i of the C_i in Figure 17, are identical to those in Figure 19, except that all of the arrows are reversed, a simple consequence of Theorem 2, and different labels might be chosen, as well.

After even a casual look at Figures 17, 18, and 19, it seems safe to say, without fear of exaggeration, that, while the control maps in Figure 17 are simple, neat, and elegant, the commutative diagrams that correspond to them in the other two figures are a mess. The first problem we notice, as suggested above, is the gross non-uniformity in the ways the commutative diagrams manage to represent composition, on the one hand (Figure 18), and class

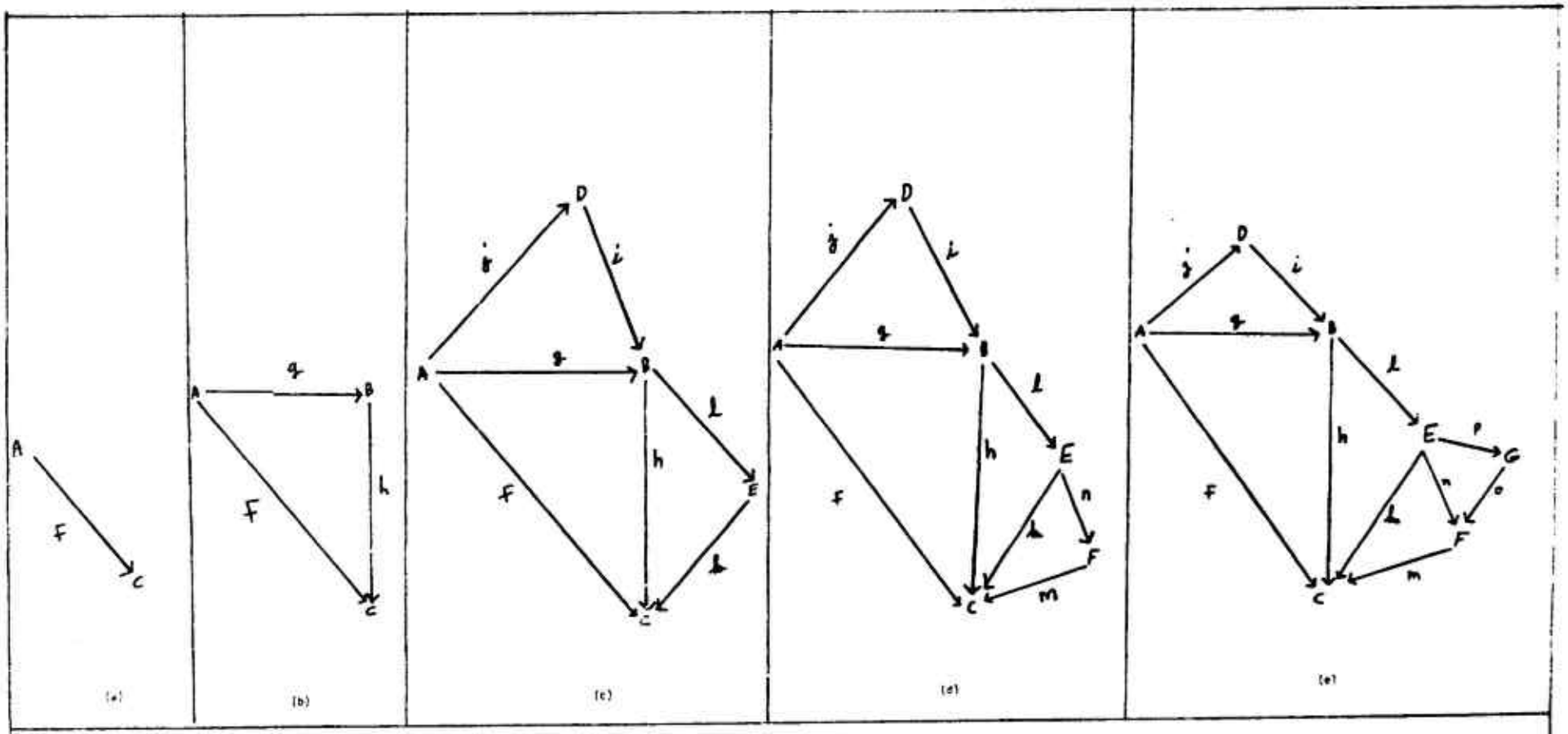


FIGURE 18: ARROW-LANGUAGE TRANSLATIONS OF THE CONTROL MAPS IN FIGURE 17 WHEN ALL CONTROL STRUCTURES ARE TAKEN TO BE JOIN (OR COMPOSITION)

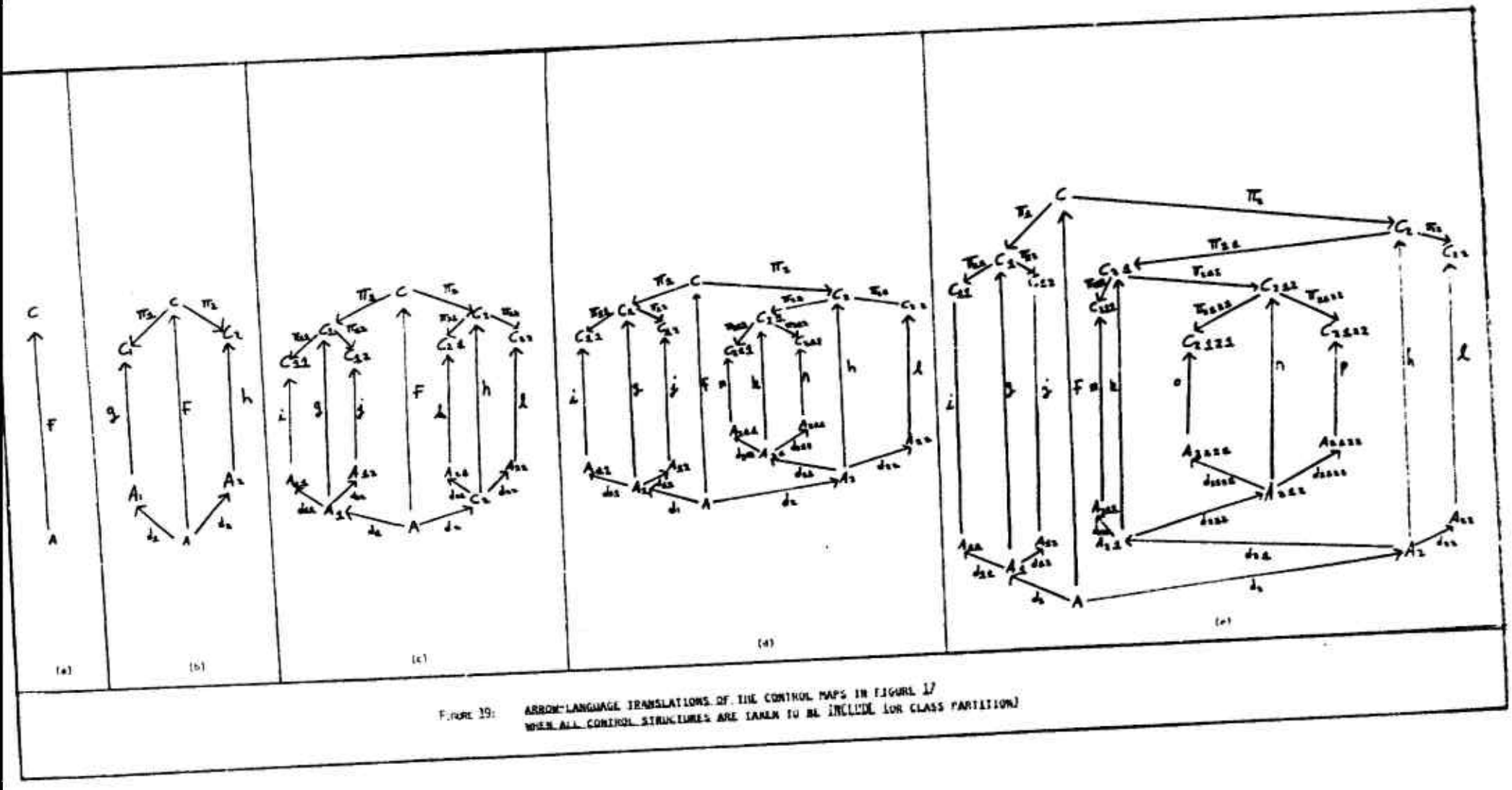
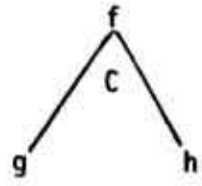


FIGURE 19: ARROW-LANGUAGE TRANSLATIONS OF THE CONTROL MAPS IN FIGURE 17 WHEN ALL CONTROL STRUCTURES ARE TAKEN TO BE INCLUDE (OR CLASS PARTITION)

partition(as well as set partition, via Theorem 2), on the other (Figure 19). What are simply two different manifestations of the single formal structure



in control map notation, with the difference indicated by straightforward specification of the data relationships involved, as seen in Section 3, become wildly unrelated kinds of formal structures when expressed as commutative diagrams. The non-uniformity of the arrow representations of composition, on the one hand, vs. the two forms of partition, on the other, is already evident in Figure 16, of course. The arrow representation of a composition includes only the mappings in the composition, for example, whereas the arrow representations of a partition (of either sort) requires two pairs of auxiliary mappings for each binary decomposition. The partition arrow diagrams, furthermore, involve a certain decompositional symmetry of the subfunctions around the parent function and geometric parallelism of the subfunctions, neither of which is in evidence at all in the arrow diagram for composition. The full extent of the complications caused by these discrepancies does not become clear, however, until we introduce more and more levels of functional structure, as in Figure 17, 18 and 19. What look in Figure 16 like interesting, but minor non-uniformities in the arrow representations of composition and partition, become a notational nightmare when we try to extend those representations beyond a single level of decomposition, as in the latter three figures.

Perhaps most damaging to the value of commutative diagrams in system design, however, is the apparently random way in which both the composition and partition diagrams expand geometrically on the page, as more levels of decomposition are introduced. As we have seen, repeated decomposition is represented quite simply and naturally for all primitive control structures in control maps by downward expansion of the tree structure, with whatever leftward and right-ward expansion is naturally involved in that. For commutative diagrams, however, the situation is not so simple, as Figures 18 and 19 clearly reveal.

For composition, shown in Figure 18, the situation is complex, but perhaps tolerable. Once we choose a direction in which to expand the initial decomposition, further expansions are all more or less in that same general direction, at least as far as the diagrams shown in that figure are concerned. Introducing more levels of decomposition could complicate the picture considerably, however, especially, for example, if we were to begin decomposing j and m. In that case we might find our diagram closing in on itself around f, unless we either expressed our further decompositions in triangles too small to read or significantly increased the size of the entire diagram.

For partitions, however, the situation is qualitatively more complex, it seems, than could ever be the case in composition. As Figure 19 reveals, repeated partition decompositions are expressed in commutative diagrams not only by outward expansion but by inward expansion, as well! Every new level of decomposition we introduce requires us to introduce a new symmetric structure like Figure 16(iib) (or c) around the arrow that represents the function we are decomposing. In decomposing h, for example, in Figure 19(b), we have to introduce l outside the diagram in Figure 19(c), automatically causing expansion in one direction, and k inside the diagram, creating pressure for expansion by the need to make room for it. In decomposing n, in contrast, in Figure 19(d), we have to introduce both o and p inside the diagram in Figure 19(e), whereas in decomposing f in Figure 19(a), we have to introduce both g and h outside the diagram in Figure 19(b). All uniformity apparently goes out the window in this case, and the diagrams have to expand arbitrarily in all directions in order just to remain intelligible as more and more decomposition takes place.

It seems fair to say that diagrams of this sort would not be a very helpful tool to the engineer in actually specifying a real system or describing a set of requirements, once the system or requirements had surpassed the most minimal level of complexity. We have not even touched on the problem, we might add, of what happens to the commutative diagrams, when the systems they represent are permitted to contain instances of more than just one of the primitive control structures. As might be expected, the complexity of the diagrams that result in that case is truly astounding, as compared to the simple and

straightforward control maps that they are functionally equivalent to, and we leave it as an enlightening exercise for the reader to investigate that fact for himself.

None of this is meant to disparage commutative diagrams, however, which have been found to be extremely useful throughout contemporary mathematics in unifying a wide range of otherwise apparently disparate phenomena (MacLane and Birkhoff, 1967; MacLane, 1972; Arbib and Manes, 1975). We ourselves used such diagrams to prove Theorem 2 (Cushing, 1978a), whose truth is not evident from control-map notation. The fact remains, however, that, however useful these diagrams may be for other purposes in other areas, they are not very useful in designing or describing either systems or requirements, nor are they particularly revealing of system structure or architecture, as is amply verified in Figures 17, 18 and 19.

Our reason for stressing this latter point is that commutative diagrams are, in fact, more revealing of system structure than are R-Nets, and therefore more useful in design as well, beyond the minimal level of complexity. As we saw in Section 2, R-Nets provide no natural mechanism for representing higher-level functional structure, which we had to indicate in Figure 4 by introducing the new notational device of drawing boxes around network subconfigurations that could be seen to be performing higher-level functions. Drawing these boxes was useful in establishing the relation between R-Nets and control maps, and thus in proving theorems 3, 4, 5, and 6, but they are not an intrinsic part of the R-Net notation itself, which does not concern itself with higher-level functional structure. One might try to augment the R-Net structure by adopting the box device as a part of an expanded notational framework, but, although this would increase the power of the framework to the level of being able to describe higher-level functional structure, the diagrams that would result from repeated decomposition would be every bit as complex as the corresponding arrow diagrams, such as those in Figures 18 and 19. This fact seems fairly clear just from looking at Figure 4 itself, and we leave any further verification of it, again, to the reader.

There is a certain similarity, in fact, between R-Net notation with boxes and commutative diagrams, and it is worth discussing that similarity briefly at this point. Speaking metaphorically, we might even say that R-Nets-with-boxes and commutative diagrams are "duals," in a very loose sense like the sense of "dual" used in projective geometry (Behnke et al., 1974), rather than the category-theoretic sense used in the formulation of Theorem 2. The interchanging of points and lines plays a role in projective geometry analogous to the reversing of the directions of arrows in category theory. Given any theorem in projective geometry formulated entirely in terms of points, lines, and incidence, we obtain another theorem by interchanging the words "point" and "line," and no further proof of a theorem obtained in this way is needed (pp. 95-96).

If, again, we let ourselves speak very loosely, rather than with the precision that led to Theorem 2, then we can see that there is a sense in which this

latter notion of duality characterizes the relation between R-Nets and arrow diagrams. R-Nets, as our Figures 1 and 2 show, use nodes (points) to represent functions and arrows (lines), in effect, to represent flowing control, whereas commutative diagrams use nodes (points) to represent data repositories, i.e., sets, and arrows (lines) to represent functions. An R-Net, in other words, will use the configuration

(1) 

to indicate that control is flowing into and then out of function f[6], whereas a commutative diagram will use the configuration

(2) 

to indicate that data flows from repository A through function f into repository B. In the R-Net the function is represented by a node and the flowing control by arrows; in the commutative diagram the function is represented by an arrow and it is the data repositories, and thus, in effect, control[7] that get represented by nodes.

There is good reason to think that this "duality" between R-Nets and commutative diagrams can be made quite precise and that a lot of interesting theoretical results could then be derived from it. At present, however, we are treating it as only a suggestive metaphor, as we have repeatedly stressed. The really important fact to observe now, in our present context, is that in both cases, the notion of control structures, except for composition, is really something quite foreign to the notation and has to be introduced, so to speak, from the outside. Composition, of course, is a triviality, since all it involves is repeated application of functions, and so can be represented simply by linking together a string of units of the forms (1) or (2). The other two control structures, however, require special configurations in both R-Nets and commutative diagrams, as we have seen.

---

[6] If f is a function, we can also view the arrows as denoting data flow into and out of the function. Data flow in any other sense is not represented, however,

[7] The "duality" of data and control is discussed more fully in Cushing (1977, 1978b). See also Cushing (1978a) and Section 3 above.

In commutative diagrams, a partition is indicated by having subfunctions run parallel to the parent function; which partition is involved is indicated by the directions of the arrows relative to the domains and ranges of the parent function. In an R-Net, in contrast, a partition is indicated by having the subfunctions "branch off" so that they have no direct geometric connection to each other, and the parent function is not shown at all; which partition is involved, furthermore, can be indicated only by introducing special ad hoc symbols ("&" and "+") to do so, since nothing otherwise already in the notation lends itself readily to that purpose, as arrow directions do in commutative diagrams. In neither case is the notion of control structure really an intrinsic part of the notation. In commutative diagrams, however, we can at least decipher what the control structures are by carefully examining the arrow directions in the various structural configurations, quite aside from what symbols happen to be added as labels to the diagrams. In R-Nets, in contrast, it is only the symbols that tell us which sort of partition is involved.

This difference is closely related to the loose "duality" between R-Nets and commutative diagrams that we have discussed. The reason that it is possible to show parent functions in a commutative diagram is precisely because functions are represented by arrows, not nodes. Given two arrows representing functions, it is a simple matter to represent their parent function as an arrow parallel and between them, as indicated in Figure 19, and higher-level functional structure then follows by repeating this process. Since functions are represented in R-Nets by nodes, however, this device is no longer available: what would it mean to make one node "parallel" to two others? The only solution it seems, is exactly the one we introduced in our translation from R-Nets to control maps, namely, drawing boxes around the nodes representing the subfunctions to get the "node" representing the parent function. One "node" of this sort containing two other nodes that represent functions is like one arrow parallel to and between two other arrows that denote functions. Our box notation is thus a natural analog for R-Nets of arrow parallelism in commutative diagrams, as well as being what seems to be the only way to get higher-level functional structure into R-Nets.

The careful reader will by now have observed that functions are also represented by nodes in control maps and will wonder why (or whether) the same deficiencies of R-Nets that we have pointed out result from this representation do not also apply to control maps. The reason why they do not is easily seen by comparing Figures 8, 9, and 10. The function nodes in a control-map tree play a very different role from that played by the function nodes in an R-Net, because the lines that connect the nodes are of a very different character in the two kinds of diagrams. The lines in an R-Net connect functions on one level to each other, possibly by way of a control structure node, whereas the lines in a control map connect functions on one level to their parent function on the next higher level. Whereas lines in an R-Net denote actual control flow through a system, the lines in a control map denote relations of control in a hierarchy of levels of control. Control map notation is based on the notion of control structure, rather than either having to introduce it artificially through special symbols or having to decipher it from the directions of arrows. As Figures 8, 9, and 10 make clear, the lines in a control map serve, in a sense, as abbreviations for the boxes in a rearranged R-Net structure with boxes. Once we start to represent things as control-map trees, however, the boxes become utterly superfluous and we are free to make full use of the control-map notation itself.

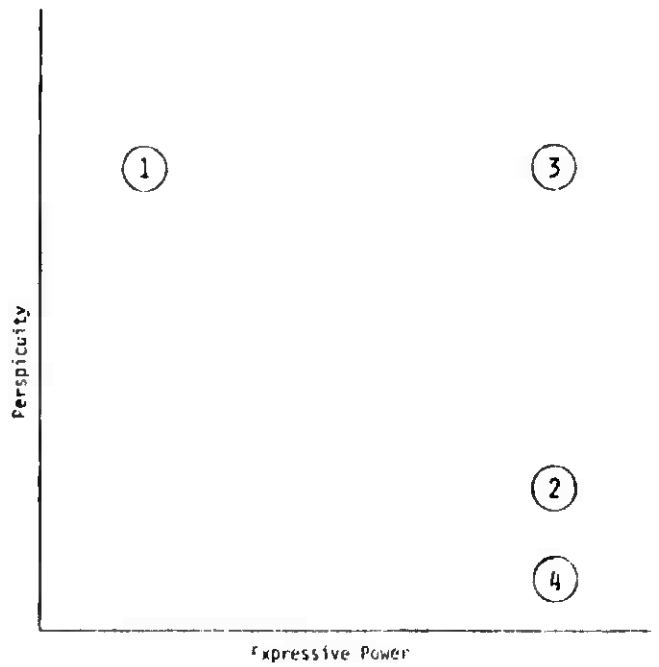
8. CONTROL MAPS, R-NETS, AND COMMUTATIVE DIAGRAMS

Our discussion to this point is summarized in Figure 20. Relative to the ranges of information they are each capable of expressing, R-Nets and control maps appear to be equally perspicuous. Both notations are straightforward to use and easy to read, providing a clear understanding of the aspects of system structure that they describe. Control maps are considerably more expressive, however, in that they naturally incorporate a way of describing higher-level functional structure, whereas R-Nets do not. The two frameworks would thus appear to be equally useful in describing small systems and simple requirements in which problems of interface correctness and the need for abstract control structures do not arise, but the use of control maps would appear to be advisable in describing larger systems and more complex requirements, in which these issues become increasingly more important.

Control maps, R-Nets-with-boxes, and commutative diagrams appear to be equal in expressive power, but differ markedly in perspicuity. All three notations are capable of expressing information about higher-level functional structure, but only control maps provide a uniform way of representing that information. Control maps, furthermore, expand in size, as complexity of decomposition increases, by a simple downward tree expansion, with no alteration to the structure of the diagram already drawn. Both R-Nets-with-boxes and commutative diagrams, however, represent increasing functional decomposition by increasing the internal complexity of the diagram, by drawing more boxes in the former case and more arrow configurations in the latter. This internal complication, in contrast to the external downward expansion of control maps, could necessitate repeated redrawings of the diagrams, as the need for larger dimensions to make room for the increasing complexity becomes evident[8]. Again, the three notational frameworks would appear to be equally suitable for describing very small systems and very simple requirements, but control maps would appear to be advisable in all other cases.

---

[8] Composition in commutative diagrams (and in R-Nets, for that matter) does expand outward (Fig. 18), but this only underscores the non-uniform character of the way those diagrams represent control structures.



1: R-Nets
2: R-Nets-with-Boxes
3: Control Maps
4: Commutative Diagrams

Figure 20: Relative Expressive Power and Perspicuity of the Notational Frameworks

BIBLIOGRAPHY

Alford, M. W. et al., "Software Requirements Engineering Methodology", SREP Final Report - Volume 1, CDRL CD05, TRW Defense and Space Systems Group, Huntsville, AL (August 1, 1977).

Arbib, Michael A. and Ernest G. Manes, Arrows, Structures, and Functors. Academic Press, New York (1975).

Behnke, H. et al. (ed.), Fundamental of Mathematics, Volume II, Geometry, MIT Press, Cambridge (1974).

Cushing, S., "The Software Security Problem and How to Solve It", Technical Report No. 6, Higher Order Software, Inc. (hereafter cited as HOS, Inc.) Cambridge, MA (July 1977).

Cushing, S., "A Note on Arrows and Control Structures: Category Theory and HOS," in "Candidate BMD Data and Axioms," Technical Report No. 15, HOS, Inc., Cambridge, MA (June 1978a).

Cushing, S. "Security Aspects of Higher Order Software," To appear in the Proceedings of the COMPSAC '78 Conference, Chicago, Nov. 1978 (IEEE Computer Soc.).

Harel, D. and R. Pankiewicz, "A Universal Flowcharter," Technical Report No. 11, HOS, Inc., Cambridge, MA (November 1977).

Hamilton, M. and S. Zeldin, "AXES Syntax Description," Technical Report No. 4, HOS, Inc., Cambridge, MA (December 1976a).

Hamilton, M. and S. Zeldin, "Integrated Software Development System/Higher Order Software Conceptual Description," Technical Report No. 3, HOS, Inc., Cambridge, MA (November 1976b).

Hamilton, M. and S. Zeldin, "Higher Order Software--A Methodology for Defining Software," IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, (March 1976c).

Hamilton, M. and S. Zeldin, "The Manager as an Abstract Systems Engineer," Digest of Papers, Fall COMPCON 77 (Washington, D.C.), IEEE Computer Society, Cat. No. 77CH1258-3C (Sept. 1977). [Technical Report No. 5, HOS, Inc., Cambridge, MA (June 1977a)].

Hamilton, M. and S. Zeldin, " Verification of an Axiomatic Requirements Specification," A Collection of Technical Papers, AIAA/NASA/IEEE/ ACM Computers in Aerospace Conference (Los Angeles, CA) (October 1977). [Technical Report No. 10, HOS, Inc., Cambridge, MA (October 1977b)].

Higher Order Software, Inc., "The Application of HOS to PLRS," HOS, Inc., TR-14, Cambridge, MA (November 1977c).

MacLane, S., Categories for the Working Mathematician, Springer-Verlag (1972).

MacLane, S. and G. Birkhoff, Algebra, Macmillan, New York (1967).